

JUDLE KNOX Wagon
NAVAL POSTGRADUATE SCHOOL
MONTREY CALIF 93940

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

AN ADAPTATION OF THE ADA LANGUAGE
FOR MACHINE GENERATED COMPILERS

by

Mark Allen Rogers
and
Linda Mary Myers

December 1980

Thesis Advisor:

B. J. MacLennan

Approved for public release; distribution unlimited

T197877

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) AN ADAPTATION OF THE ADA LANGUAGE FOR MACHINE GENERATED COMPILERS		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis: December 1980
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Mark Allen Rogers Linda Mary Myers		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December 1980
		13. NUMBER OF PAGES 207
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Ada, Compiler, LEX, YACC, Scanner, Parser		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The Ada language has been designated by the Department of Defense to replace the computer languages currently in use by the various services for tactical computer programs. This thesis modifies the Cornell Subset of Ada so that it is suitable for producing a LALR(1) grammar. Machine generated compiler tools such as LEX and YACC, available under the UNIX operating system, are then used to implement the scanner and the parser for this subset of Ada.		

Approved for public release; distribution unlimited

An Adaptation of the Ada Language
For Machine Generated Compilers

by

Mark A. Rogers
Lieutenant Commander, ¹¹United States Navy
B.S., United States Naval Academy, 1970

and

Linda M. Myers
Lieutenant, United States Navy
B.S., State University of New York at Albany, 1972

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1980

ABSTRACT

The Ada language has been designated by the Department of Defense to replace the computer languages currently in use by the various services for tactical computer programs. This thesis modifies the Cornell Subset of Ada so that it is suitable for producing a LALR(1) grammar. Machine generated compiling tools such as LEX and YACC, available under the UNIX operating system, are then used to implement the scanner and the parser for this subset of Ada.

TABLE OF CONTENTS

I.	INTRODUCTION -----	6
	A. BACKGROUND -----	6
	B. APPROACH -----	6
II.	THE COMPLETE ADA GRAMMAR -----	8
	A. HISTORY AND DESCRIPTION OF ADA -----	8
	B. YACC AND LEX -----	12
III.	UNIX TOOLS -----	17
	A. PREFACE -----	17
	B. SPECIFICATIONS FOR USING YACC -----	18
	C. SPECIFICATIONS FOR USING LEX -----	24
IV.	THE CORNELL SUBSET OF ADA -----	29
	A. ADA/CS (CORNELL SUBSET) -----	29
	B. ADA/CS ERRORS -----	30
	C. ADA/CS COMPARED TO THE COMPLETE ADA LANGUAGE -----	32
	D. ABBREVIATION OF THE SUBSET -----	34
V.	THE MODIFIED CORNELL SUBSET (ADA/MCS) -----	35
	A. PREFACE -----	35
	B. CHANGING ADA/CS TO ADA/MCS -----	35
	C. OBSERVATIONS -----	45
VI.	TEST PROGRAMS AND RESULTS -----	46
VII.	CONCLUSIONS -----	48
	A. DOCUMENTATION INADEQUACIES -----	48
	B. FUTURE SYMBOL TABLE IMPLEMENTATION -----	49

C.	INCLUSION OF ADA/MCS IN THE FULL ADA GRAMMAR -----	50
D.	SUMMARY -----	51
APPENDIX A	- ADA.LEX -----	53
APPENDIX B	- ADA/MCS -----	58
APPENDIX C	- PARSE TABLE -----	83
APPENDIX D	- TEST PROGRAMS AND OUTPUTS -----	167
LIST OF REFERENCES	-----	206
INITIAL DISTRIBUTION LIST	-----	207

I. INTRODUCTION

A. BACKGROUND

The Ada¹ language has been designated by the Department of Defense to replace the computer languages currently in use by the various services for tactical computer programs. Because of this designation the Ada language will become an increasingly important language for computer science sub-specialists in the Navy. For this reason the Ada language was chosen as a broad topic for this thesis, and specifically to begin the work necessary for the eventual realization of an Ada compiler for the Naval Postgraduate School.

B. APPROACH

As a first thesis project using the Ada language it was decided to utilize the UNIX² compiler generator tools, LEX and YACC (described in detail in Section Two), to produce a three-address code intermediate language. This approach was taken to allow future thesis projects to utilize the

¹Named in honor of Ada Augusta, Lady Lovelace, the daughter of the poet, Lord Byron, and Charles Babbage's programmer.

²UNIX is a Trademark/Service Mark of the Bell System, and is an operating system for the PDP-11 at the Naval Postgraduate School.

"front end" compiler thus produced for adaptations to the various computer systems in use at the Naval Postgraduate School.

It was discovered that the Ada language grammar is not defined in a format which is easily adaptable to machine generated compilers, thus necessitating the eventual thrust of this thesis, the adaptation of the Ada language for machine generated compilers.

In Section Two the complete Ada grammar is introduced. Section Three describes the use of the UNIX tools YACC and LEX. Section Four fully describes the Cornell Subset of Ada, followed by a description of the Modified Cornell Subset (Ada/MCS) in Section Five. Section Six introduces the test programs and the results obtained and the conclusions of this thesis are presented in Section Seven.

II. THE COMPLETE ADA GRAMMAR

A. HISTORY AND DESCRIPTION OF ADA

The language Ada is a direct result of the Department of Defense High Order Language Commonality program which began in 1975 with the goal of establishing a single high order computer programming language appropriate for DOD embedded computer systems. The Ada language was designed with three major objectives: program reliability and maintenance, a concern for programming as a human activity, and efficiency.

The Ada programming language has strong expressive power designed to cover a wide application domain. It is a modern algorithmic language designed to satisfy the Steelman³ requirements. A brief description of the language, along with an excellent language summary from the "Reference Manual For the Ada Programming Language Proposed Standard Document" [Ref. 2] which was received during the final stages of this thesis, is quoted in the following pages.

³DoD requirements for the common high order language were formalized in a series of documents extensively reviewed by the Services, industrial organizations, universities, and foreign military departments which culminated in the Steelman Report to which Ada language has been designed.

An Ada program is composed of one or more program units, which can be compiled separately. Program units may be subprograms (which define executable algorithms), packages (which define collections of entities), or tasks (which define concurrent computations). Each unit normally consists of two parts: a specification, containing the information that must be visible to other units, and a body, containing the implementation details, which need not be visible to other units.

This distinction of the specification and body, and the ability to compile units separately allow a program to be designed, written, and tested as a set of largely independent software components.

An Ada program will normally make use of a library of program units of general utility. The language provides means whereby individual organizations can construct their own libraries. To allow accurate control of program maintenance, the test of a separately compiled program unit must name the library units it requires.

Program units:

A subprogram is the basic unit for expressing an algorithm. There are two kinds of subprograms: procedures and functions. A procedure is the logical counterpart to a series of actions. For example, it may read in data, update variables, or produce some output. It may have parameters, to provide a controlled means of passing information between the procedure and the point of call. A function is the logical counterpart of the computation of a value. It is similar to a procedure, but in addition will return a result.

A package is the basic unit for defining a collection of logically related entities. For example, a package can be used to define a common pool of data and types, a collection of related subprograms, or a set of type declarations and associated operations. Portions of a package can be hidden from the user, thus allowing access only to the logical properties expressed by the package specification.

A task is the basic unit for defining a sequence of actions that may be executed in parallel with other similar units. Parallel tasks may be implemented on multi-computers, multiprocessors, or with interleaved execution of a single processor. A task unit may define either a single executing task object or a task type defining similar task object.

Declarations and Statements:

The body of a program unit generally contains two parts: a declarative part, which defines the logical entities to be used in the program unit, and a sequence of statements, which defines the execution of the program unit.

The declarative part associates names with declared entities. For example, a name may denote a type, a constant, a variable, or an exception. A declarative part also introduces the names and parameters of other nested subprograms, packages, and tasks to be used in the program unit.

The sequence of statements describes a sequence of actions that are to be performed. The statements are executed in succession (unless an exit, return, or goto statement, or the raising of an exception causes execution to continue from another place).

An assignment statement changes the value of a variable. A procedure call invokes execution of a procedure after associating any arguments provided at the call with the corresponding formal parameters of the subprogram.

Case statements and if statements allow the selection of an enclosed sequence of statements based on the value of an expression or on the value of a condition.

The basic iterative mechanism in the language is the loop statement. A loop statement specifies that a sequence of statements is to be executed repeatedly until an iteration clause is completed or an exit statement is encountered.

A block comprises a sequence of statements preceded by the declaration of local entities used by the statements.

Certain statements are only applicable to tasks. A delay statement delays the execution of a task for a specified duration. An entry call is written as a procedure call; it specifies that the task issuing the call is ready for a rendezvous with another task that has this entry. The called task is ready to accept the entry call when its execution reaches a corresponding accept statement, which specifies the actions then to be performed. After completion of the rendezvous, both the calling task and the task having the entry may continue their execution in parallel. A select statement allows a selective wait for one of several alternative rendezvous. Other forms of the select statement allow conditional or timed entry calls.

Execution of a program unit may lead to exceptional situations in which normal program execution cannot continue. For example, an arithmetic computation may exceed the maximum allowed value of a number, or an attempt may be made to access an array component by using an incorrect index value. To deal with these situations, the statements of a program unit can be textually followed by exception handlers describing the actions to be taken when the exceptional situation arises. Exceptions can be raised explicitly by a raise statement.

Data Types:

Every object in the language has a type which characterizes a set of values and a set of applicable operations. There are four classes of types: scalar types (comprising enumeration and numeric types), composite types, access types, and private types.

An enumeration type defines an ordered set of distinct enumeration literals, for example a list of states or an alphabet of characters. The enumeration types BOOLEAN and CHARACTER are predefined.

Numeric types provide a means of performing exact or approximate computations. Exact computations use integer types, which denote sets of consecutive integers. Approximate computations use either fixed point types, with absolute bound on the error, or floating point types, with relative bound on the error. The numeric types INTEGER and REAL are predefined.

Composite types allow definitions of structured objects with related components. The composite types in the language provide for arrays and records. An array is an object with indexed components of the same type. A record is an object with named components of possibly different types.

A record may have distinguished components called discriminants. Alternative record structures that depend on the values of discriminants can be defined within a record type.

Access types allow the construction of linked data structures created by the execution of allocators. They allow several variables of an access type to designate the same object, and components of one object to designate the same or other objects. Both the elements in such a linked data structure and their relation to other elements can be altered during program execution.

Private types can be defined in a package that conceals irrelevant structural details. Only the logically necessary properties (including any discriminants) are made visible to the users of such types.

The concept of a type is refined by the concept of a subtype, whereby a user can constrain the set of allowed values in a type. Subtypes can be used to define subranges of scalar types, arrays with a limited set of index values, and records and private types with particular discriminant values.

Other Facilities:

Representation specifications can be used to specify the mapping between data types and features of an underlying machine. For example, the user can specify that objects of a given type must be represented with a specified number of bits, or that the components of a record are to be represented in a specified storage layout. Other features allow the controlled use of low level, non portable, or implementation dependent aspects, including the direct insertion of machine code.

Input - output is defined in the language by means of predefined library packages. Facilities are provided for input - output of values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided.

Finally the language provides a powerful means of parameterization of program units, called generic program units. The generic parameters can be types and subprograms (as well as objects) and so allow general algorithms to be applied to all types of a given class.

B. YACC AND LEX

Initial research for this thesis began with Ada Syntax Summary which is presented in the "Preliminary Ada Reference Manual" [Ref. 1]. The initial goal of producing a parse table for the complete Ada grammar involved preparation and modification of the syntax to be able to use the PDP-11 facilities of Yet Another Compiler-Comiler (YACC) [Ref. 5]

and LEX [Ref. 7]. YACC is a program which takes the syntactic (BNF) grammar rules of the language and generates a program which takes as its input the output of a scanner program (tokens) and parses a program written in the defined language (i.e. Ada). YACC also allows semantic rules to be defined and executed for each syntactic construct of the language, thereby completing the compilation process. In place of an input stream of tokens, YACC can also use the scanner routine generated by the LEX program as a subroutine to provide the input tokens.

LEX is a program that takes as its input regular expressions defining the characteristics of the raw character strings of a language and generates a scanner routine. This scanner routine has the capability to eliminate blanks and comments and to output groups of characters (tokens) which are meaningful to the syntactic rules of the language.

Executed together LEX and YACC produce a "machine generated" compiler whose input is the character string of the program to be compiled, and whose output can be either an intermediate code (generalized form of execution actions), or a machine executable code.

Using YACC for the Ada language was a two-step process since the Ada language specification is written in a modified Extended Backus-Naur Form (EBNF) and YACC requires the language grammar to be expressed in Backus-Naur Form (BNF). Therefore, step one was conversion of EBNF to BNF and step two was actually executing the resulting BNF grammar using YACC and LEX.

The most significant difference between the EBNF and BNF forms (other than the fact that EBNF is far more convenient for human reading of a grammar) is that two sets of metasymbols must be removed from the EBNF grammar to produce an equivalent BNF grammar. These are the square brackets [...] meaning zero or one occurrences, and the brackets {...} meaning zero or more repetitions; also, several symbols must be replaced in the EBNF grammar to make the productions acceptable to YACC. The replacement operator, double-colon-equal (:=), must be colon (:). All trivial terminals (parentheses, semicolons, commas, etc.) must be enclosed in single quotes. All other nonterminals must be explicitly indicated to YACC and, finally, the head symbol production rule must be the top rule.

The symbol replacement, explicit nonterminals and head symbol rule placement must all be accomplished manually. The brackets mentioned previously are automatically removed via a conversion process which yields new production rules with new nonterminals. These new nonterminals are formed by concatenating the original nonterminals with prefixes such as "fst." and "opt." [Ref. 8]. The entire conversion process, along with the details of executing the resulting BNF grammar on YACC are presented in [Ref. 8].

Once the complete Ada syntax listed in [Ref. 1] was modified to the proper BNF form, the grammar was examined

in detail for errors. Two errors were found in Appendix E of the Preliminary Ada Reference Manual [Ref. 1]. First, the token "character←literal" was used in the production rule:

```
enumeration←literal ::= identifier | character←literal.
```

"Character←literal" does not appear on the left hand side of any production and was, therefore, changed to

"character←string". Secondly, in the "accept←statement" rule, in the nonterminal "entry←name", the word "entry" should have appeared in italics indicating it was simply a syntactic modifier and not part of the token "name".

After these errors were corrected, the complete Ada grammar in appropriate BNF form acceptable to YACC was executed. The input to YACC did not initially include any lexical analyzer or action code.

The sheer length of the resulting BNF form of the Ada grammar caused a memory overload on the PDP-11 and YACC could not produce a complete parse table. This problem was solved by manually producing a list of every nonterminal in the full Ada grammar and sorting the list according to length (some nonterminals were embedded within others). Using the UNIX line editor, each nonterminal was then replaced by two letter sequences, i.e., aa, ab, ac, ... The BNF grammar thus produced was acceptable to YACC and required far less memory space. YACC then produced a complete parse table for the abbreviated version of the full Ada grammar in proper BNF form.

The resulting parse table was riddled with shift/reduce and reduce/reduce conflicts (almost fivehundred total conflicts). After verification that the grammar was an exact duplicate of the syntax summary presented in the "Preliminary Ada Reference Manual" [Ref. 1], with the exeption of the two minor errors discussed previously, review of this attempted parsing of the grammar verified that the Ada language as defined in the Preliminary Manual was not LALR(1). With the exception of a final execution of the full Ada grammar through YACC incorporating the changes made to a subset of the grammar (discussed in Section Seven), this completed the work on the full Ada grammar.

III. UNIX TOOLS

A. PREFACE

This section of the thesis is intended to familiarize the reader with the various language development tools available under the UNIX Operating System and to supplement available documentation that often glosses over the intricacies of language development using UNIX. This section will also provide specific guidance to follow-on thesis projects that use UNIX for the purposes of automatic compiler generation.

The compiler process of taking an input program written in a source language and producing as output an equivalent program in a target language is commonly subdivided into five distinct phases: lexical analysis (scanning), syntax analyzing (parsing), semantic analysis, code optimization and code generation. The tools available under UNIX concentrate on the automation of only the scanning and parsing phases of compilation. Automation of the remaining phases has yet to become a reality.

Although an overview of LEX [Ref. 7] and YACC [Ref. 5] was presented in Section Two, it was slanted towards a description of these UNIX tools rather than an explanation of how they are specifically used to build a compiler. This section provides detailed procedures for starting with the

given syntax of a strong subset of Ada (Ada/MCS) and automatically generating a lexical analyzer and parse tables.

B. SPECIFICATIONS FOR USING YACC

Ada/MCS is in Extended Backus-Naur-Form (EBNF). As previously mentioned, EBNF is very convenient for human description of a grammar but is not acceptable to the YACC system. Therefore, the conversion program [Ref. 8] introduced in Section Two, which is stored in the Naval Post-graduate School Computer Sciences Laboratory under the name "ebnftobnf", must be used to convert the subset in EBNF to BNF. The program "ebnftobnf" is a YACC program input and must be processed with the command:

```
% yacc ebnftobnf .
```

The percentage symbol is the UNIX prompt symbol. The operator is expected to enter an operating system command. The execution of the above command produces a file called "y.tab.c" that is written in the programming language C [Ref. 9]. This C program must be compiled with the YACC library by using the following operating system level command:

```
% cc y.tab.c -ly .
```

The output of this compilation is the desired EBNF to BNF conversion program and is called "a.out". It is recommended that "a.out" be changed to a different name to remind the user that it is the EBNF to BNF conversion program. The

conversion program object code file used throughout this thesis is called "ebnf.conv". The preceding steps to arrive at a conversion program must only be accomplished once. After "ebnf.conv" is formed by the steps outlined above, it may be used to convert any EBNF grammar to BNF, once the symbols referred to in Section Two (double-colon-equals and trivial terminals) have been properly replaced or designated.

After forming the "ebnf.conv" program, Ada/MCS was converted to BNF form using the following UNIX operating system command:

```
% ebnf.conv <ada.ebnf.sub >ada.bnf.sub .
```

The "<" symbol means use the given file name ("ada.ebnf.sub") for input and the ">" designates that the output file be called "ada.bnf.sub".

After "ada.bnf.sub" has been created, several steps must be taken before executing the BNF form of the subset with YACC. Using the UNIX line editor ("ed ada.bnf.sub"), the command:

```
> g/↑$/d
```

was used to remove all blank lines from the file. The ">" character is the UNIX line editor prompt and is not entered by the user. This decreased the size of the file and was a preliminary step in matching up the line numbers of the

BNF grammar with the referenced reduction numbers from the output of YACC.

Throughout the Ada grammar, the underscore was used extensively to more explicitly describe names (as in `array_type_definition`). Since the terminals at the Naval Postgraduate School are not equipped with the underscore character, it was replaced with the character "`+`". YACC uses "`+`" to indicate the position of the parsing process (which token is currently being analyzed) so it was decided to change this character, to eliminate confusion, with the line editor command:

```
> g/\+/s//\$g .
```

The "\$" character was not used for the underscore in the original EBNF form of the grammar since the `ebnf` to `bnf` conversion program expects "`_`" or "`+`" as name separators for its input, and will not accept a "\$". Since all punctuation must be enclosed in single quotes, a problem occurred when the single quote in the production:

```
predefined+attribute ::= name ' identifier
```

had to be enclosed in a single quotes. The resulting form in the EBNF subset became:

```
predefined+attri : name ' ' id .
```

This was converted to:

```
predefined+attri : name ' ' ' id
```


in the BNF form. In using YACC, as in C, the backslash ("\") is an escape character within the literals and therefore the line editor command:

```
> s/' ' '\ ' /
```

must be issued to change the line that "name ' ' id" appeared in to "name '\ ' id".

The final change to "ada.bnf.sub" while in the line editor was to ensure that the head production appeared as the first production in the BNF form of the subset. This was accomplished by using the line editor move command ("m") to move the appropriate lines to the start of the grammar.

After completion of the above steps, "ada.bnf.sub" was in an acceptable form for YACC, but several steps were still required to actually use YACC. The next step in the process was to ensure the line numbers for each production in the BNF grammar corresponded to the reduction numbers in the parse tables in "y.output", the output file obtained from a proper execution of YACC. This was most efficiently accomplished by switching UNIX line editors to vi [Ref. 6]. Vi (visual) is a display oriented interactive text editor that was best suited to combine lines in productions that did not have a null (empty) option. The BNF version of a grammar that is produced via the conversion program discussed above left a blank line following every production, whether it had a null option or not. The Vi command "J"

was used to close-up the first options of every production that did not have a null option, so that every production option (including null) would have a line number that corresponded to the reduction numbers in each state of the parse tables of "y.output".

The whole purpose of the above procedures was to produce a set of production rules acceptable to YACC, and thus be able to build a compiler that can process a program in Ada to produce either a "yes" or "no" answer as to the program's syntactic correctness or to compile it to some target language. To accomplish this goal the BNF grammar must be put into a UNIX file that fits the YACC input format:

```
token list
%%
BNF rules
%%
programs
```

The token list was entered into a file called "ada.tokens", and a small file was created called "ada.includelex" which consisted of two lines

```
%%
# include "lex.yy.c" (the output of Lex -,
                      discussed later).
```

Thus, the operating system level command:

`% cat ada.tokens ada.bnf.sub ada.includelex>ada.temp`
concatenates the required files into a file called "ada.temp" which can be processed by YACC using the operating system level command:

```
% yacc -v ada.temp .
```

The "-v" option produces a verbose output file called "y.output" which is a complete parse table listing. Also produced by this command is a file called "y.tab.c" which is a compilable C source program. If YACC is executed without the verbose option, no "y.output" is created.

C program statements for processing the grammar into a target language as each production is recognized can be mixed into the BNF rules using the " = {...}" format to enclose each action. For example:

```
decl : obj$decl
      = { printf(" | decl \n") ; }
      | type$decl
      = { printf(" | decl \n") ; } ; .
```

These action statements were not added to the grammar until after the BNF rules themselves were determined to be LALR(1) (no parsing errors). For this thesis, the actions that have been added to the grammar only output the parsing actions as they are accomplished. In follow-on theses these actions can be changed to code generation actions.

Following the creation of a "y.output" file, the UNIX operating system command move ("mv") was used to change the file name as each execution of YACC will produce a new version of "y.output". At this point, several pages of output were saved by editing the new "y.output" and using the line editor command:

```
> g/↑$/d
```

to delete all blank lines.

The complete process described above to make changes in an EBNF grammar and then follow these changes through to a final parse table listing was used several times to finally ensure that the Ada subset (Ada/MCS) was a LALR(1) grammar. The details of this procedure are discussed fully in Section Five.

C. SPECIFICATIONS FOR USING LEX

The YACC program output "y.tab.c" makes a subroutine call to "yylex()". This call is a request for input tokens from a scanning routine. This scanner routine is produced by the LEX program. As described in Section Two the LEX program inputs are regular expressions and C language action statements. Appendix A is a listing of "ada.lex", the input to the LEX program for Ada/MCS.

The general format of the LEX input is:

```
[definitions]
```

```
%%
```



```
[rules]
```

```
%%
```

```
[user subroutines]
```

In the definitions section of "ada.lex" are the regular expressions in the format of:

```
name      space  regular expression  .
```

For example:

```
l  [a-zA-Z]
```

```
d  [0-9]
```

```
c  ( {l} | {d} )
```

defines "l" (for letter) as a character from a-z or A-Z, "d" (for digit) as a character 0-9, and "c" (for character) as a letter or a digit.

The specific meanings of all the special characters used to form the regular expression are given in the LEX manual [Ref. 7].

Also in the definitions section are changes to the internal array sizes for the LEX program that override the default conditions set by the original program. For the Ada language it was necessary to change the array sizes.

The rules section is in the format:

```
lexical definition      space  C action statement  .
```

Any reserved word can be returned by the LEX scanning routines as a separate token value by using the word as a lexical definition, instead of returning every word as an

identifier. This saves the compiler writer from checking each identifier with a reserved word table in the parser section of the compiler and allows this evaluation to be completed in the scanner.

Rules are included in this section for eliminating blanks, tabs, and new lines "[\t\u]", and eliminating comments "\-\\-[\$\n]*". These rules do not return any values to the parser. All other rules return a token value via the "return ();" C statement.

The single special characters (i.e. =) could have been left out of the rules sections and would have been given default values. A decision was made to supply token values to these characters for easier verification of the correctness of the tokens returned to the parser by this scanner.

The scanner program produced by LEX may be run as a scanner only when a "main ()" routine is included in the user subroutines. This was done as a stand alone test of the scanner (see Section Six for results). The separate compilation is done with the operating system command sequence:

```
% lex ada.lex
% cc lex.yy.c -ll -ls
```

The first command executes the LEX program with "ada.lex" as the input file, and produces an output file "lex.yy.c" which is a C compilable file.

The last command compiles "lex.yy.c" with the LEX library routines to form a C object program of the scanner.

The operating system command:

```
% a.out <ada.prog >output
```

executes the scanner thus produced, with an Ada language test program, and outputs a list of the tokens parsed. This command may be repeated for any number of test programs.

The user subroutine section is not necessary for the scanner to interface with the parser produced by the YACC program. The "main()" subroutine in this section is therefore commented out in the C format: "/* ... */", before the parser and scanner are combined.

The parser produced by YACC and the scanner produced by LEX may be combined by the following operating system command sequence:

```
% lex ada.lex
% yacc ada.temp
% cc y.tab.c -ly -ll -ls
% mv a.out ac .
```

The first command is the same as executing the LEX program for a separate compiler. It used "ada.lex" as input and produces "lex.yy.c".

The second command executes YACC with "ada.temp" as input and produces "y.tab.c" as output. Again, the "-v" for a verbose output is optional.

The third command compiles "y.tab.c" (which contains `#include "lex.yy.c"`) along with the YACC, LEX and UNIX operating system libraries. It should be noted that in order for "y.tab.c" to be compiled properly it had to be slightly modified. Since each grammar production became part of a huge "switch" statement in the C language, the length of the grammar exceeded the option limit (128) in the "switch" statement. The entire "switch" was simply divided into four switch statements of equal length and combined using the "if then else" construct. The compilation of "y.tab.c" produces the compiler object program "a.out" which in the last command is renamed "ac" (for Ada compiler).

The operating system command:

```
% ac <ada.prog >output
```

executes the compiler with an Ada language test program.

This thesis project concludes with a verification of the combined actions of the parser and scanner and the output of this last command is a trace of the scanning and parsing actions (see Section Six for results).

All of the output produced by "ac" is preliminary information for verification purposes and can easily be deleted by eliminating all the "printf" C commands in the "ada.lex" and "ada.bnf.sub" files.

IV. THE CORNELL SUBSET OF ADA

A. ADA/CS (CORNELL SUBSET)

The intention of this thesis was to begin the process of building a compiler starting with a strong subset of Ada and allowing additional capabilities and features of the language to be added to this basic building block. This approach was taken due to time constraints and the size of the Ada language.

In November of 1979 the Department of Computer Science at Cornell University issued Technical Report TR79-395, "Ada/CS -- An Instructional Subset of the Programming Language Ada" [Ref. 4] which was used as the starting point for the language syntax for a machine generated front end compiler currently under production.

The designers of Ada were faced with clear injunctions against any definitions of subsets of the complete Ada programming language. Those restrictions were to ensure standardization of the language and perhaps "to pressure the designers to be frugal in their proposals by denying them the escape that certain exotic features would be ignored in practical subsets."⁴

⁴ Archer, J., "Ada/CS -- An Instructional Subset of the Programming Language Ada", Technical Report TR79-395, Computer Science Department, Cornell University.

When the design phase of Ada was completed, the originators of the Cornell Subset of Ada saw no reason to continue the restrictions against defining a subset of Ada, and Viewed its development as a necessary instructional tool that would allow programmers the freedom to not use certain language features of Ada. Another concern of the subset writers was that every programming language that is generally accepted as viable (and DoD assures us that Ada will be accepted) is subjected to subsetting. Thus, to avoid a myriad of incompatible subsets, the authors of Ada/CS formulated their subset as the basis for a small number of precisely defined standard subsets. The intent of Ada/CS was strictly for introductory-level programming instruction and the authors felt a strong need to overcome the mostly political use of FORTRAN for basic instructional purposes.

The advantages of a formal subset of Ada are significant; especially in relation to compilers. Subset compilers are easier and less expensive to develop, faster in operation, and executable on smaller systems. In addition, a compiler can provide much more effective diagnostics if it knows that certain language features will not be used.

B. ADA/CS ERRORS

Ada/CS was defined by removing some constructs entirely from the full language and limiting some that remained. Regardless, the syntax summary presented in Ada/CS was

supposedly a proper subset of the Ada syntax presented in "The Preliminary Ada Reference Manual" and all productions of the subset were purportedly created by simply removing phrases from the reference version. Although this was generally the case, the following errors were noted and corrected in the Ada/CS specification.

In the "basic←loop" production, a semi-colon did not follow the optional identifier following "END LOOP". There were two cases where the zero or one symbol [...] had been replaced by the zero or more symbol {...} in the subset. These two cases were in the "relation" production, around "relational←operator simple←expression" and in the "body" production around "visibility restriction". The last error was another case of losing the italic designation of a syntactic modifier for the nonterminal "name" (there was a different italics error discussed in Section Two that was in the full Ada grammar). In the production for "type←mark", in the nonterminal "type←name", the work "type" should have appeared in italics to demonstrate its syntactic modifier status. The corrected version of the above errors are in Appendix B under "Ada/CS". Note that italicized words are simply eliminated in production rules.

Although the above errors were a bit tedious to discover, they were easily corrected and not considered significant. One major shortcoming of Ada/CS which was very significant from a compiler writer's point of view was the

absence of a head production in the grammar. A head production is defined as the one grammar rule which all other grammar rules will eventually reduce to. The head production rule when recognized by the parser, signals that the end of the input character stream has been reached and the compilation is finished. This problem resulted from the deletion of the production "compilation ::= {compilation←unit}" from the subset. Whether or not the authors of Ada/CS intended to eliminate separate compilation of individual units from the subset is not clear. It can be inferred by the omission of the "compilation" production that this capability was not desired in Ada/CS. But, by this omission, the head production was also eliminated. The subset was therefore expanded to include the "compilation" start symbol and associated productions from the Ada reference manual.

C. ADA/CS COMPARED TO THE COMPLETE ADA LANGUAGE

Several features of the complete Ada language were omitted in the CS subset. The main features of the complete Ada language were discussed in the language summary portion of Section Two. The major features that were not included in the Cornell Subset are discussed below.

The facilities for defining tasks and all concurrent programming features have been eliminated in the subset. Control over variable record representation and access variables has been lost. The name overloading or redefinition

capability has been discarded along with user-defined generic procedures. The last feature omitted was the exception mechanism.

Along with these major construct omissions, further compression of the full Ada language was accomplished by attaching semantic and syntactic restrictions to the constructs included in Ada/CS. The principal restrictions were imposed on expressions, subprograms, declarations, case statements and goto statements. Expressions were modified by not allowing array or slice expressions and discarding exponentiation from the subset. The subprogram construct was modified considerably to greatly simplify the language. Subprograms cannot be used to redefine operators. Also, actual parameters corresponding to out and in out formal parameters must represent locations which can be legally assigned values of the appropriate type. Formal parameter array subscript range values cannot be specified. Procedure side-effects are not allowed in expressions; only function subprograms can return values. The last subprogram restriction was that keyword parameter mechanism were not allowed in Ada/CS. Declarations were limited so that no anonymous types were allowed. Case statements were slightly modified by demanding that at least two choices be specified and requiring a "when others" clause if all legal cases are not syntactically guaranteed.

The last restriction was allowing the goto statement to transfer control forward only. The Ada/CS Technical Report [Ref. 4] provides further details of each construct restriction.

D. ABBREVIATION OF THE SUBSET

Section Two discussed the difficulties encountered with the full Ada grammar, in EBNF form, in relation to available memory space on the PDP-11. Since the final subset used in this thesis consisted of approximately half the productions of the complete Ada language, the somewhat drastic abbreviation mechanism of converting every nonterminal to a two letter sequence was not necessary in the subset. The subset productions were therefore abbreviated in a logical manner and the resulting BNF form is easily understood. Appendix B lists the modified Cornell subset (Ada/MCS) along with a listing of "ada.sub.terms" that may be used to clarify any abbreviation that might be ambiguous to the reader.

The details of Ada/MCS are discussed in the next section, along with a complete explanation of how Ada/CS was modified to be LALR(1).

V. THE MODIFIED CORNELL SUBSET (ADA/MCS)

A. PREFACE

As stated in Section Three, YACC requires its input grammar to be LALR(1). The original Cornell Subset (Ada/CS) was not LALR(1) and when executed with YACC resulted in over two-hundred reduce/reduce and shift/reduce conflicts. Reference [3] provides a complete explanation of these conflicts. In the absence of any user directives to the contrary, YACC invokes two disambiguating rules to deal with conflicts. In a shift/reduce conflict, the default is to do the shift. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule in the input sequence. Therefore, although it is possible for YACC to produce a parser even in the presence of some conflicts, it was one of the prime goals of this thesis to rewrite the Cornell Subset of Ada so that it was LALR(1) and therefore the resulting parse table would be free of all conflicts.

B. CHANGING ADA/CS TO ADA/MCS

Following the correction of the grammar errors discussed in Section Three, various changes to the BNF form of the Ada subset were made. Each conflict in the initial parse table for Ada/CS was carefully examined to discover the source of the conflict and a revision of the appropriate

production(s) was attempted so that the language would not be restricted by the revision. This was not possible in all cases and some revisions caused minor limitations to the subset grammar. Several conflicts in the parser automatically generated by YACC could only be eliminated by restricting the Ada grammar.

This section details each change to the Ada/CS production rules that was made in order to change it to LALR(1). Unless specifically stated, the change was not restrictive to the grammar.

The first change was a separation of the sections of Ada/CS that rightfully belonged in the lexical analyzer. Reference [4] contained section 2.3 through 2.5 that defined identifiers, numbers and the character string which were in turn deleted from Ada/MCS and defined more appropriately in "ada.lex" (see Appendix A).

After adding the head production "compilation←unit" and its associated productions as detailed in Section Three, it was determined that several reduce/reduce conflicts were being caused by Ada/CS not distinguishing whether an "id" was meant to be part of an "enum←lit" or part of a more generalized expression or part of "name". The following productions illustrate this confusion:

```
enum←lit : id | char←string
```

```
name : id | indexed←comp | selected←comp | predefined←attr
```



```

lit : num | enum←lit | char←string
pri : lit | var | subprog←call | aggr
      | qualified←expr | (expr)  .

```

The production:

```

lit : num | enum←lit | char←string

```

was allowing "id" to come through the literal production above into the "primary", "factor", "term", "expression" chain which reduced "id" into expressions without going through the "name" production. The conflict thus created meant that an "id" which was part of a "enum←lit" was reduced into one of the "expression" structures. For example, in the "enum←type←defn":

```

TYPE light IS (red, amber, green);

```

The identifier "red" may first be reduced to a name, where in fact it is strictly an enumeration literal. This somewhat intricate problem was corrected by simply eliminating "enum←lit" from the "lit" production, resulting in:

```

literal : num | char←string  .

```

The productions:

```

pri : lit | aggr | var | subprog←call
      | qualified←expr | '(' expr ')'
var : name opt.←.disc←range.←.
      | name '.' ALL

```



```

opt.+disc+range.+ : ε
    | '(' disc+range ')'
subprog call : name opt.+para+assoc..lst.+para+assoc..+.
    | '(' para+assoc fst.+para+assoc')'

```

led to confusion as to which optional null production was being parsed when null options (ϵ) were encountered. For example, the subprogram call:

```
print+report;
```

with no parameters could be interpreted by the parser as a variable.

The solution was to eliminate the "opt.+disc+range" and the "opt.+para+assoc..lst.+para+assoc..+." in the above productions for "var" and "subprog+call" so that these productions would be parsed only if they began with " name '(' " and if there was no trailing parenthesis the parsing table would be left with just a "name" which was added to the primary production above to yield the following corrected productions which will parse the same language:

```

pri : lit | aggr | var | subprog+call
    | qualified+expr | '(' expr ')' | name
var : name '.' ALL
    | name '(' disc+range ')'
subprog+call : name '(' para+assoc+fist.+para+assoc.' )' .

```

A problem occurred with the "expr" production in that if no option followed a "rel" in the production:


```

expr : rel fst.AND.rel.
      | rel fst.OR.rel.
      | rel fst.XOR.rel.

```

the parse table did not know which empty option to parse.
For example, the program segment:

```

IF a<b THEN
    LOOP ...

```

is an example of an expression consisting of only one relation which would lead to the compiler not being able to determine which option of this production is being reduced.

The solution was formulated by changing "expr" to encompass the logical operator functions in the following manner:

```

expr : expr log+op rel
      | rel      .

```

At this point in the conversation from Ada/CS to the LALR(1) Ada/MCS, it became apparent that changing the BNF form of the grammar was much more difficult than analyzing the EBNF form and making changes to it. The procedure therefore changed at this point in the thesis to strictly working with the EBNF grammar.

Another conflict was caused by "expr" being a legal case of the "aggr" production:

```

aggr : '(' comp+assoc { ',' comp+assoc } ')'

```



```

comp+assoc : choice { '|' choice } '=>'  expr ;
pri : lit | aggr | var | subprog+call
      | qualified+expr | '(' expr ')' | name .

```

An example from the language illustrates this point. The expression

```
(a<b)
```

parses to either an "aggr" or an "expr" and the real determination of the type of language construct desired must be accomplished semantically.

The solution was to eliminate " '(' expr ')' " from the "pri" production which also contained "aggr", and let " '(' expr ')' " parse through "aggr". The difference in code generation between an "expr" and an "aggr" (an array or record value) must be decided at the semantic level.

Another significant ambiguity occurred in Ada/CS due to the fact that it allowed so many productions to begin with "simple+expr". The production

```
range : simple+expr '..' simple+expr
```

led to the most of the parsing conflicts and was therefore changed by limiting the first "simple+expr" to an "id" or a "lit" as follows:

```

range : range+id '..' simple+expr
range+id : id | lit .

```


Perhaps the major source of parsing errors in Ada/CS concerning the fact that "name '(' " could be the start of an indexed+comp (array), a "var" (part of an array), a "subprog+call", or a "qualified+expr". The difference was not recognizable in the compiler because all the above productions used the same delimiter, i.e., " () ". For example, when a subprogram call has parameters as in

```
cos (x,y);
```

this call is syntactically the same as an array variable

```
an+array (x,y);
```

The solution to this delimiter situation was two-fold. The first part was to remove "qualified+expr" completely from the new grammar and to let the semantic actions tell whether " name (expr) " was a qualified expression or a subprog+call. The second phase of the solution was to make " name '(' " a general form for a subprogram, array variable or qualified expression. This would make it necessary for the parser action statements to differentiate using the symbol table and generate errors where new incorrect possibilities could be parsed. This step also allowed the productions "para+assoc" and "actual+para" to be deleted. The resulting productions from the above language restrictions follow:

```
name : id | subprog+array+var | selected+comp  
      | predefined+attri
```



```

selected←comp : name '.' id
predefined←attri : name ''' id
var : name '.' ALL
subprog←array←var : name '(' expr { ',' expr } ')' .

```

In the productions:

```

disc←range : [type←mark RANGE] range
aggr : '(' comp←assoc { ',' comp←assoc } ')'
comp←assoc : [choice { '|' choice } `=>'] expr
choice : simple←expr | disc←range | OTHERS
simple←expr : [unary←op] term { adding←op term }

```

when an "aggr" was parsed through the "comp←assoc" and "choice" productions, a decision was faced whether to parse to a "simple←expr" or a "disc←range". These productions had different optional beginnings and the parser could not choose which production it was attempting to parse, as can be seen in the following example:

```

C : TABLE := (0..4 => 0);
C : TABLE := (5..9 => 1.0);

```

where the first numbers (0 and 5) are the beginnings of the discrete range, but the compiler cannot distinguish until the ".." is reached whether the numbers might be the beginnings of simple expressions.

It was decided to allow only one of the two productions to have an optional beginning; therefore, the language

was restricted by eliminating the option of a null production at the beginning of the "disc range" production:

```
disc←range : type←mark RANGE range .
```

In every instance that "AND" or "OR" was encountered, a conflict was generated when the parser did not know whether to parse to "cond" or "expr". The following productions illustrate the problem that a LALR(1) parser had parsing the grammar as written because "AND THEN" or "OR ELSE" created a necessity to look ahead two tokens:

```
cond : expr { AND THEN expr }  
      | expr { OR ELSE expr }  
expr : expr log←op rel | rel  
log←op : AND | OR | XOR .
```

Ada/MCS removed this (look ahead by two tokens) conflict by changing the "cond" production so that "AND THEN" and "OR ELSE" became single tokens:

```
cond : expr { AND.THEN expr }  
      | expr { OR.ELSE expr } .
```

This production in turn created an ambiguity since if neither "AND.THEN" or "OR.ELSE" were present, the parser did not, again, know which empty option to follow. The empty options were combined as follows:

```
cond : expr { cond←ext }  
cond←ext : AND.THEN expr  
          | OR.ELSE expr
```


thus creating a "condition extension" production, which did not change or restrict the language.

Another series of conflicts were created by the parser's attempts to decide whether the "choice" option was present or not and therefore distinguishing whether "id" and "lit" were starting a "choice" or starting an "expr" in the productions:

```
range←id : id | lit
choice : range←id | disc←range | OTHERS
comp←assoc : choice { '|' choice } '=>' expr .
```

These conflicts were solved by creating a restricted choice ("restr←choice") in the "comp←assoc" production as follows:

```
comp←assoc : restr←choice { '|' choice } '=>' expr
              | pri [{ '|' choice } '=>' expr]
restr←choice : disc←range | OTHERS
choice : range←id | restr←choice
```

thus restricting the grammar by eliminating the option of allowing a "simple←expr" to start a "choice". Note that the second option for "comp←assoc" used "pri" instead of "simple←expr". This further restricts Ada/MCS in that a unary operator must be put in parentheses when used in the "comp←assoc" production. This restriction was a valid trade-off rather than allowing "comp←assoc" to begin with "simple←expr" which would have caused further complicated

changes to Ada/MCS that would considerably lengthen the EBNF form of the grammar.

C. OBSERVATIONS

The preceding description of the various changes made to Ada/CS in order to modify the grammar to be LALR(1) are traditional examples of the trade-offs that language implementors must face: restricting the grammar or lengthening the parse table. Since most compilers have specific time constraints that must be met, each restriction must be evaluated to intrinsically measure the time saved by shortening the parse table versus the intended capability of the original language grammar.

Each modification that was made to Ada/CS was the final result of many experiments in efficiency and the resulting Ada/MCS does not lose much capability compared with Ada/CS. Appendix B contains the Ada/MCS grammar and Appendix C is the conflict-free parse table that was automatically created by YACC for Ada/MCS.

VI. TEST PROGRAMS AND RESULTS

Appendix D contains several test programs, written in Ada/MCS, that make use of the majority of the features available in the subset. The test programs are not elaborate but serve to illustrate that the automatic scanner and parser developed for Ada/MCS functions properly. Each program illustrates different variations of syntactic constructs from Ada/MCS. The output of each compiled Ada/MCS program is an easily decipherable scanner-parser action trace. This trace is provided in Appendix D for the first four programs only to demonstrate the format of the scanner-parser action trace; the other syntactic constructs tested are shown without trace results.

Each test program has two versions: a syntactically correct form and an incorrect form. The format of the scanner-parser action trace is:

token number (from ada.lex)		token
		reduction
		reduction .

These reductions continue until another token is needed. Another token number from ada.lex is issued and the process continues until the head production is reached. For example, the program excerpt "PROCEDURE test IS" would have a

scanner-parser action trace as follows:

```
28  | PROCEDURE
    | opt.vis$restriction.
    | opt.SEPARATE
    | subprog$nature
54  | test
    | designator
18  | IS
    | opt.formal$part.
    | opt.RETURN.type$mark
    | subprog$spec  .
```

If a syntax error occurs, a line is skipped in the scanner-parser actions trace and YACC issues the statement "syntax error". The error recovery procedure defined in the YACC action statements cause the parser to request tokens from the scanner until an end of statement character (;) is received. The printer prints the message " | Parse error ", and the procedure continues to parse beginning with the next statement. Ada.lex, along with deleting comments and ignoring blanks, tabs, and newlines, also issues the diagnostic "Scanner error : unknown symbol" in every instance that an unknown character is encountered. The character is then ignored and the parse continues.

The scanner-parse action traces in Appendix D illustrate the above actions.

VII. CONCLUSIONS

A. DOCUMENTATION INADEQUACIES

The original intention that ultimately led to this thesis was a full implementation of a compiler for the programming language Ada. As is common with most compiler writing projects, unforeseen problems were encountered; specifically, inadequate documentation of the UNIX system automatic compiler generation tools and ambiguities in the Ada language. These problems forced a redirection of the thesis goals to the task of laying the groundwork for a future implementation of Ada through the development of the first two steps of the compiler process: the scanner and the parser.

The documentation for YACC and LEX had no specific errors, but several weaknesses existed. The documentation did not provide enough detailed explanation of LEX or YACC or specific examples for a first-time user of these tools. It also did not include any specific implementation details concerning the Naval Postgraduate School PDP-11/50 system. For example, the LEX documentation listed two distinct methods of specifying character constants: quotation marks and backslashes; however, on the NPS system the only method implemented was the backslash.

Section Three of this thesis was an attempt to alleviate these documentation shortcomings by providing a detailed description of the use of these UNIX tools for the project of writing an Ada compiler (or any compiler).

Another serious documentation deficiency was the inclusion in the UNIX documentation of an introductory description of a program called EYACC (Extended Yet Another Compiler-Compiler) which purported to expand the memory capability of the YACC program that was eventually used for this thesis. Due to the size of the Ada grammar, EYACC was originally chosen for this thesis to take advantage of its expanded memory capabilities. EYACC was discarded at the later stages of the thesis because it was discovered that it was only partially implemented on the NPS PDP-11/50 system. It would produce a parse table listing, but when compiled, the listing would yield no output.

B. FUTURE SYMBOL TABLE IMPLEMENTATION

The major objective of this thesis was to verify the possibility of a machine generated parser and scanner for Ada. In the process of revising the Ada grammar to form an LALR(1) grammar it was discovered that some ambiguities in the language were due to the similarity of the syntactic constructs for different semantic actions. For example, the array variable " array←a(x,y,z) " is syntactically similar to the subprogram call "cos(x,y)". This problem could be

resolved in either of two ways: (1) modify the scanner to use a symbol table lookup procedure and return a token value for "array+a" or "cos" that is more specific than an identifier (e.g., "array+id" and "subprog+id"), or (2) make the grammar rule recognize the general syntactic form, (i.e., "identifier (expression, expression ...)") and use a symbol table lookup procedure in the action portion of the grammar rule to distinguish between "array+a" and "cos".

Due to the time constraints of this thesis it was not possible to implement a symbol table, consequently it was decided to delay the symbol table lookup procedure as explained in the second alternative given above. This allowed for the verification of the scanner-parser interface without use of a symbol table.

This solution was decided upon for the ambiguities discovered between array or record names and subprogram names, and for the ambiguity between array or record parameters and variables.

When a symbol table is eventually incorporated as a follow-on to this thesis, it may be desirable to use a symbol table lookup procedure in the scanner, resulting in simpler semantic actions.

C. INCLUSION OF ADA/MCS IN THE FULL ADA GRAMMAR

The elimination of the shift/reduce and reduce/reduce errors in the Ada/MCS subset discussed in Section Five made it desirable to investigate the effect of the Ada/MCS

changes on the full Ada grammar. This investigation was accomplished by manually changing the abbreviated BNF form of the full grammar and processing the modified full grammar with YACC.

This process decreased the parsing errors from 490 to 286.

D. SUMMARY

This thesis was written using the "Preliminary Ada Reference Manual" as its primary source of documentation and, as has been discussed, it was discovered that Ada is not easily implemented by automated compiler generation tools. During the final stages of this thesis, the proposed standard reference manual for Ada was acquired and a review of this document revealed that the original ambiguities of the language still exist in the proposed final version of Ada.

One conclusion of this thesis is the fact that a programmer may use Ada to express his ideas without ambiguity, but a reader may find the same program ambiguous. The difficulties discussed in Section Five concerning the "name (" dilemma very clearly substantiate this point. Variable names are not easily distinguishable by the reader without referring to the data declarations section. It is acknowledged that the designers of the Ada language intended arrays and subprogram calls to look alike, since

it may sometimes be desirable to implement arrays as sub-programs (for example, sparse arrays). However, it is felt that the readability problem created by this intended ambiguity may eventually create maintenance difficulties for ongoing software development projects.

Section Five provides a detailed analysis of the changes that were necessary to convert an existing subset of Ada to a LALR(1) grammar. In the development process of Ada the primary consideration seems to have been an attempt to create a language that would impose a structured programming environment upon the users, for the enhancement of the software engineering process of program development, without enough consideration being given to the language implementors. Perhaps the strongest conclusion drawn from this thesis is that Ada compilers may be more costly and take longer to develop than anticipated due to these language ambiguities. This could result in delaying the replacement of the current DOD tactical languages in use with a standard acceptable Ada language.

APPENDIX A

Ada.lex

This appendix is a listing of the input file to the LEX program and shows the reserved words, and regular expressions defining identifiers, numbers, and character strings.

```
l      [a-zA-Z]
d      [0-9]
c      ({l}|{d})
cx     (\+|{l}|{d})
dx     (\+|{d})
i      ({d}{dx}*)
idecex ({i}\.{i}{e})
idec   ({i}\.{i})
iex    ({i}{e})
b      ({i}\#({l}|{d})+(\+|{l}|{d})*)
e      (\E((\+?{i})|(\-{i})))
charstr  \"[\40-\176]*\"
id      {l}{cx}*
num     {i}|{b}|{idec}|{idecex}|{iex}
%e      600
%p      2000
```



```

%n                300
%a                2000
%o                2000
%%

ALL      {printf("1 : ");ECHO;printf(" \n");return(1);}
AND      {printf("2 : ");ECHO;printf(" \n");return(2);}
ARRAY    {printf("3 : ");ECHO;printf(" \n");return(3);}
ASSERT   {printf("4 : ");ECHO;printf(" \n");return(4);}
BEGIN    {printf("5 : ");ECHO;printf(" \n");return(5);}
BODY     {printf("6 : ");ECHO;printf(" \n");return(6);}
CASE     {printf("7 : ");ECHO;printf(" \n");return(7);}
CONSTANT {printf("8 : ");ECHO;printf(" \n");return(8);}
ELSE     {printf("9 : ");ECHO;printf(" \n");return(9);}
ELSIF    {printf("10 : ");ECHO;printf(" \n");return(10);}
END      {printf("11 : ");ECHO;printf(" \n");return(11);}
EXIT     {printf("12 : ");ECHO;printf(" \n");return(12);}
FOR      {printf("13 : ");ECHO;printf(" \n");return(13);}
FUNCTION {printf("14 : ");ECHO;printf(" \n");return(14);}
GOTO     {printf("15 : ");ECHO;printf(" \n");return(15);}
IF       {printf("16 : ");ECHO;printf(" \n");return(16);}
IN       {printf("17 : ");ECHO;printf(" \n");return(17);}
IS       {printf("18 : ");ECHO;printf(" \n");return(18);}
LOOP     {printf("19 : ");ECHO;printf(" \n");return(19);}
MOD      {printf("20 : ");ECHO;printf(" \n");return(20);}

```



```

NOT      {printf("21 ; ");ECHO;printf(" \n");return(21);}
NULL     {printf("22 ; ");ECHO;printf(" \n");return(22);}
OF       {printf("23 ; ");ECHO;printf(" \n");return(23);}
OR       {printf("24 ; ");ECHO;printf(" \n");return(24);}
OTHERS   {printf("25 ; ");ECHO;printf(" \n");return(25);}
OUT      {printf("26 ; ");ECHO;printf(" \n");return(26);}
PACKAGE  {printf("27 ; ");ECHO;printf(" \n");return(27);}
PROCEDURE {printf("28 ; ");ECHO;printf(" \n");return(28);}
RANGE    {printf("29 ; ");ECHO;printf(" \n");return(29);}
RECORD   {printf("30 ; ");ECHO;printf(" \n");return(30);}
RESTRICTED {printf("31 ; ");ECHO;printf(" \n");return(31);}
RETURN   {printf("32 ; ");ECHO;printf(" \n");return(32);}
REVERSE  {printf("33 ; ");ECHO;printf(" \n");return(33);}
SEPARATE {printf("34 ; ");ECHO;printf(" \n");return(34);}
THEN     {printf("35 ; ");ECHO;printf(" \n");return(35);}
TYPE     {printf("36 ; ");ECHO;printf(" \n");return(36);}
USE      {printf("37 ; ");ECHO;printf(" \n");return(37);}
WHEN     {printf("39 ; ");ECHO;printf(" \n");return(39);}
WHILE    {printf("50 ; ");ECHO;printf(" \n");return(50);}
XOR      {printf("51 ; ");ECHO;printf(" \n");return(51);}
AND.THEN {printf("52 ; ");ECHO;printf(" \n");return(52);}
OR.ELSE  {printf("53 ; ");ECHO;printf(" \n");return(53);}
\:\?    {printf("70 ; ");ECHO;printf(" \n");return(70);}
\\.     {printf("71 ; ");ECHO;printf(" \n");return(71);}

```



```

=\>      {printf("72 ; ");ECHO;printf(" \n");return(72);}
\/\?     {printf("73 ; ");ECHO;printf(" \n");return(73);}
\<\=     {printf("74 ; ");ECHO;printf(" \n");return(74);}
\>\=     {printf("75 ; ");ECHO;printf(" \n");return(75);}
\<\<     {printf("76 ; ");ECHO;printf(" \n");return(76);}
\>\>     {printf("77 ; ");ECHO;printf(" \n");return(77);}
\;        {printf("78 ; ");ECHO;printf(" \n");return(78);}
\  
(        {printf("79 ; ");ECHO;printf(" \n");return(79);}
\  
)        {printf("80 ; ");ECHO;printf(" \n");return(80);}
\!        {printf("81 ; ");ECHO;printf(" \n");return(81);}
\,        {printf("82 ; ");ECHO;printf(" \n");return(82);}
\.        {printf("83 ; ");ECHO;printf(" \n");return(83);}
\'        {printf("84 ; ");ECHO;printf(" \n");return(84);}
\=        {printf("85 ; ");ECHO;printf(" \n");return(85);}
\<        {printf("86 ; ");ECHO;printf(" \n");return(86);}
\>        {printf("87 ; ");ECHO;printf(" \n");return(87);}
\&        {printf("88 ; ");ECHO;printf(" \n");return(88);}
\*        {printf("89 ; ");ECHO;printf(" \n");return(89);}
\/        {printf("90 ; ");ECHO;printf(" \n");return(90);}
\:        {printf("91 ; ");ECHO;printf(" \n");return(91);}
\+        {printf("92 ; ");ECHO;printf(" \n");return(92);}
\-        {printf("93 ; ");ECHO;printf(" \n");return(93);}
{id}      {printf("54 ; ");ECHO;printf(" \n");return(54);}
{num}     {printf("55 ; ");ECHO;printf(" \n");return(55);}

```



```

-\-[\f$\n]*      ;      /*delete comments*/

{charstr} {printf("56 | ");ECHO;printf(" \n");return(56);}

[ \t\n]          ;      /*ignore blanks,tab,newline*/

%%

/*

main(){

    int nextoken;

    nextoken = 1;

    while (nextoken != 0){

        nextoken = yylex();

    }

}

*/

```


APPENDIX B

Ada/MCS

The appendix is a listing of the four files that define Ada/MCS.

Ada.terms.mean

This portion of Appendix B is simply a listing of all the abbreviations used in forming the subset Ada/MCS. This file is not an input to YACC.

aggr :aggregate
approx :approximate
assoc :association
attri :attribute
char :character
chars :characters
comp :component
cond :condition
constr :constraint
decl :declaration
defn :definition
disc :discrete
enum :enumeration

exp :exponent
expr :expression
ext :extention
fac :factor
id :identifier
int :integer
iter :iteration
lit :literal
log :logical
mod :module
mult :multiplying
num :number
obj :object
op :operator
para :parameter
pri :primary
prog :program
rel :relation
relal :relational
restr :restricted
seq :sequence
spec :specification
stmt :statement
stmts :statements

str :string

var :variable

vis :visibility

Ada.tokens Ada/MCS Ada.includelex

These three files are concatenated to form the input to YACC. This input file provides YACC with the token values that will be returned from the scanner, the BNF grammar rules, and the actions to be performed by the parser.

%token	'\$'	0
%token	ALL	1
%token	AND	2
%token	ARRAY	3
%token	ASSERT	4
%token	BEGIN	5
%token	BODY	6
%token	CASE	7
%token	CONSTANT	8
%token	ELSE	9
%token	ELSIF	10
%token	END	11
%token	EXIT	12
%token	FOR	13
%token	FUNCTION	14
%token	GOTO	15
%token	IF	16
%token	IN	17

%token	IS	18
%token	LOOP	19
%token	MOD	20
%token	NOT	21
%token	NULL	22
%token	OF	23
%token	OR	24
%token	OTHERS	25
%token	OUT	26
%token	PACKAGE	27
%token	PROCEDURE	28
%token	RANGE	29
%token	RECORD	30
%token	RESTRICTED	31
%token	RETURN	32
%token	REVERSE	33
%token	SEPARATE	34
%token	THEN	35
%token	TYPE	36
%token	USE	37
%token	WHEN	39
%token	WHILE	50
%token	XOR	51
%token	AND.THEN	52

%token	OR.ELSE	53
%token	id	54
%token	num	55
%token	char\$str	56
%token	':='	70
%token	'..'	71
%token	'=>'	72
%token	'/='	73
%token	'<='	74
%token	'>='	75
%token	'<<'	76
%token	'>>'	77
%token	';'	78
%token	'('	79
%token	')'	80
%token	' '	81
%token	','	82
%token	'.'	83
%token	'\''	84
%token	'='	85
%token	'<'	86
%token	'>'	87
%token	'&'	88
%token	'*'	89

%token	'/'	90
%token	':'	91
%token	'+'	92
%token	'-'	93

%%

```

compilation$unit:      opt.vis$restriction.      opt.SEPARATE.
unit$body

```

```

      = {printf("  | compilation$unit\n");} ;

```

```

opt.vis$restriction.:

```

```

      = {printf("  | opt.vis$restriction.\n");}

```

```

    | vis$restriction

```

```

      = {printf("  | opt.vis$restriction.\n");} ;

```

```

opt.SEPARATE.:

```

```

      = {printf("  | opt.SEPARATE. \n");}

```

```

    | SEPARATE

```

```

      = {printf("  | opt.SEPARATE. \n");} ;

```

```

decl: obj$decl

```

```

      = {printf("  | decl \n");}

```

```

    | type$decl

```

```

      = {printf("  | decl \n");}

```

```

    | error ';' = {printf("  | Parse error\n");} ;

```

```

opt.CONSTANT.:

```

```

      = {printf("  | opt.CONSTANT. \n");}

```

```

    | CONSTANT

```



```

        = {printf("    | opt.CONSTANT. \n");} ;

opt.$expr.:
        = {printf("    | opt.$expr. \n");}
    | ':' expr
        = {printf("    | opt.$expr \n");} ;

obj$decl: id$list ':' opt.CONSTANT. type opt.$expr. ';'
        = {printf("    | obj$decl \n");} ;

fst.$id.:
        = {printf("    | fst.$id. \n");}
    | fst.$id. ',' id
        = {printf("    | fst.$id. \n");} ;

id$list: id fst.$id.
        = {printf("    | id$list \n");} ;

type: type$mark
        = {printf("    | type \n");} ;

type$defn: enum$type$defn
        = {printf("    | type$defn \n");}
    | int$type$defn
        = {printf("    | type$defn \n");}
    | array$type$defn
        = {printf("    | type$defn \n");}
    | record$type$defn
        = {printf("    | type$defn \n");} ;

type$mark: name

```



```

        = {printf("    ; type$mark \n");} ;

constr: range$constr

        = {printf("    ; constr \n");} ;

type$decl: TYPE id IS type$defn ';'

        = {printf("    ; type$decl \n");} ;

range$constr: RANGE range

        = {printf("    ; range$constr \n");} ;

range: range$id '..' simple$expr

        = {printf("    ; range \n");} ;

range$id: id

        = {printf("    ; range$id \n");}

        ; lit

        = {printf("    ; range$id \n");} ;

fst.enum$lit.:

        = {printf("    ; fst.enum$lit. \n");}

        ; fst.enum$lit. enum$lit

        = {printf("    ; fst.enum$lit. \n");} ;

enum$type$defn: '(' enum$lit ',' fst.enum$lit. ')'

        = {printf("    ; enum$type$defn \n");} ;

enum$lit: id

        = {printf("    ; enum$lit \n");}

        ; char$str

        = {printf("    ; enum$lit \n");} ;

int$type$defn: range$constr

```



```

        = {printf("    | int$type$defn \n");} ;

fst.$index.:
        = {printf("    | fst.$index. \n");}
        | fst.$index. ',' index
        = {printf("    | fst.$index. \n");} ;

array$type$defn: ARRAY '(' index fst.$index. ')' OF
type$mark
        = {printf("    | array$type$defn \n");} ;

index: disc$range
        = {printf("    | index \n");}
        | type$mark
        = {printf("    | index \n");} ;

disc$range: type$mark RANGE range
        = {printf("    | disc$range \n");} ;

fst.$comp$assoc.:
        = {printf("    | fst.comp$assoc. \n");}
        | fst.$comp$assoc. ',' comp$assoc
        = {printf("    | fst.$comp$assoc. \n");} ;

aggr: '(' comp$assoc fst.$comp$assoc. ')'
        = {printf("    | aggr \n");} ;

fst.$choice.:
        = {printf("    | fst.$choice. \n");}
        | fst.$choice. '!' choice
        = {printf("    | fst.$choice. \n");} ;

```



```

opt..1st.$choice..$.expr.:
    ? {printf("      | opt..1st.$choice..$.expr.
\n");}

    | fst.$choice. '=>' expr
      = {printf("      | opt..1st.$choice..$.expr.
\n");} ;

comp$assoc: restr$choice  fst.$choice. '=>' expr
    = {printf("      | comp$assoc \n");}

    | pri  opt..1st.$choice..$.expr.
      = {printf("      | comp$assoc \n");} ;

restr$choice: disc$range
    = {printf("      | restr$choice \n");}

    | OTHERS
      = {printf("      | restr$choice \n");} ;

choice: range$id
    = {printf("      | choice \n");}

    | restr$choice
      = {printf("      | choice \n");} ;

record$type$defn: RECORD  como$list  END  RECORD
    = {printf("      | record$type$defn \n");} ;

fst.obj$decl.:
    = {printf("      | fst.obj$decl. \n");}

    | fst.obj$decl. obj$decl
      = {printf("      | fst.obj$decl. \n");} ;

```



```

comp$list: fst.obj$decl.

        = {printf("  | comp$list \n");} ;

name: id

        = {printf("  | name \n");}

! subprog$array$var

        = {printf("  | name \n");}

! selected$comp

        = {printf("  | name \n");}

! predefined$attri

        = {printf("  | name \n");} ;

selected$comp: name '.' id

        = {printf("  | selected$comp \n");} ;

predefined$attri: name '\\' id

        = {printf("  | predefined$attri \n");} ;

lit: num

        = {printf("  | lit \n");}

! char$str

        = {printf("  | lit \n");} ;

var: name '.' ALL

        = {printf("  | var \n");} ;

fst.$expr.:

        = {printf("  | fst.$expr. \n");}

! fst.$expr. ',' expr

        = {printf("  | fst.$expr. \n");} ;

```



```

subprog$array$var: name '(' expr fst.$expr. ')'
                    = {printf(" | subprog$array$var \n");} ;

opt.relal$op.simple$expr.:
                    = {printf(" | opt.relal$op.simple$expr.
\n");}

                    | relal$op simple$expr
                    = {printf(" | opt.relal$op.simple$expr.
\n");} ;

opt.NOT.:
                    = {printf(" | opt.NOT. \n");}

                    | NOT
                    = {printf(" | opt.NOT. \n");} ;

opt.constr.:
                    = {printf(" | opt.constr. \n");}

                    | constr
                    = {printf(" | opt.constr. \n");} ;

rel: simple$expr opt.relal$op.simple$expr.
    = {printf(" | rel \n");}

    | simple$expr opt.NOT. IN range
    = {printf(" | rel \n");}

    | simple$expr opt.NOT. IN type$mark opt.constr.
    = {printf(" | rel \n");} ;

expr: expr log$op rel
     = {printf(" | expr \n");}

```



```

| rel
    = {printf(" | expr \n");};

opt.unary$op.:
    = {printf(" | opt.unary$op. \n");}

| unary$op
    = {printf(" | opt.unary$op. \n");} ;

fst.adding$op.term.:
    = {printf(" | fst.adding$op.term. \n");}

| fst.adding$op.term. adding$op term
    = {printf(" | fst.adding$op.term. \n");} ;

simple$expr: opt.unary$op. term fst.adding$op.term.
    = {printf(" | simple$expr \n");} ;

fst.mult$op.fac.:
    = {printf(" | fst.mult$op.fac. \n");}

| fst.mult$op.fac. mult$op fac
    = {printf(" | fst.mult$op.fac. \n");} ;

term: fac fst.mult$op.fac.
    = {printf(" | term \n");} ;

fac: pri
    = {printf(" | fac \n");} ;

pri: lit
    = {printf(" | pri \n");}

| aggr
    = {printf(" | pri \n");}

```



```

| name
    = {printf(" | pri \n");}

| var
    = {printf(" | pri \n");} ;

log$op: AND
    = {printf(" | log$op \n");}

| OR
    = {printf(" | log$op \n");}

| XOR
    = {printf(" | log$op \n");} ;

relal$op: '='
    = {printf(" | relal$op \n");}

| '/='
    = {printf(" | relal$op \n");}

| '<'
    = {printf(" | relal$op \n");}

| '<='
    = {printf(" | relal$op \n");}

| '>'
    = {printf(" | relal$op \n");}

| '>='
    = {printf(" | relal$op \n");} ;

adding$op: '+'
    = {printf(" | adding$op \n");} ;

```



```

| '-'
    = {printf(" | adding$op  \n");}

| '&'
    = {printf(" | adding$op  \n");} ;

unary$op: '+'
    = {printf(" |  unary$op  \n");}

| '-'
    = {printf(" |  unary$op  \n");}

| NOT
    = {printf(" |  unary$op  \n");} ;

mult$op: '*'
    = {printf(" |  mult$op  \n");}

| '/'
    = {printf(" |  mult$op  \n");}

| MOD
    = {printf(" |  mult$op  \n");} ;

fst.stmt.:
    = {printf(" |  fst.stmt.  \n");}

| fst.stmt. stmt
    = {printf(" |  fst.stmt.  \n");} ;

seq$of$stmts: fst.stmt.
    = {printf(" |  seq$of$stmts  \n");} ;

stmt: simple$stmt
    = {printf(" |  stmt  \n");} ;

```



```

| compound$stmt
    = {printf(" | stmt \n");}

| '<<' id '>>' stmt
    = {printf(" | stmt \n");}

| error ';' = {printf("Parse error\n");} ;

simple$stmt: assignment$stmt
    = {printf(" | simple$stmt \n");}

| subprog$call$stmt
    = {printf(" | simple$stmt \n");}

| exit$stmt
    = {printf(" | simple$stmt \n");}

| return$stmt
    = {printf(" | simple$stmt \n");}

| goto$stmt
    = {printf(" | simple$stmt \n");}

| assert$stmt
    = {printf(" | simple$stmt \n");}

| NULL ';'
    = {printf(" | simple$stmt \n");} ;

compound$stmt: if$stmt
    = {printf(" | compound$stmt \n");}

| case$stmt
    = {printf(" | compound$stmt \n");}

| loop$stmt

```



```

        = {printf("  | compound$stmt  \n");} ;
assignment$stmt: var  ':= '  expr  ';'
        = {printf("  | assignment$stmt  \n");}
      | name  ':= '  expr  ';'
        = {printf("  | assignment$stmt  \n");} ;
subprog$call$stmt: name  ';'
        = {printf("  | subprog$call$stmt  \n");} ;
opt.expr.:
        = {printf("  | opt.expr.  \n");}
      | expr
        = {printf("  | opt.expr.  \n");} ;
return$stmt: RETURN  opt.expr.  ';'
        = {printf("  | return$stmt  \n");} ;
fst.ELIF.cond.THEN.seq$of$stmts.:
        =
          {printf("
fst.ELIF.cond.THEN.seq$of$stmts.  \n");}
          |  fst.ELIF.cond.THEN.seq$of$stmts.  ELIF  cond
THEN  seq$of$stmts
          =
          {printf("
fst.ELIF.cond.THEN.seq$of$stmts.  \n");} ;
opt.ELSE.seq$of$stmts.:
        =  {printf("  |  opt.ELSE.seq$of$stmts.
\n");}
          |  ELSE  seq$of$stmts

```



```

                = {printf("      |      opt.ELSE.seq$of$stmts.
\n"));} ;

if$stmt:      IF      cond      THEN      seq$of$stmts
fst.ELIF.cond.THEN.seq$of$stmts.      opt.ELSE.seq$of$stmts.
END IF ';'

                = {printf("      |      if$stmt \n"));} ;

fst.cond$ext.:

                = {printf("      |      fst.cond$ext. \n"));}
| fst.cond$ext. cond$ext
                = {printf("      |      fst.cond$ext. \n"));} ;

cond: expr  fst.cond$ext.

                = {printf("      |      cond \n"));} ;

cond$ext: AND.THEN  expr

                = {printf("      |      cond$ext \n"));}
| OR.ELSE  expr

                = {printf("      |      cond$ext \n"));} ;

fst.WHEN.choice..1st.$.choice..$.seq$of$stmts.:

                = {printf("      |      fst.WHEN.choice..1st.$.choice..$.seq$of$stmts. \n"));}
|      fst.WHEN.choice..1st.$.choice..$.seq$of$stmts.

WHEN  choice  fst.$.choice.  '=>'  seq$of$stmts

                = {printf("      |      fst.WHEN.choice..1st.$.choice..$.seq$of$stmts. \n"));} ;

case$stmt:      CASE      expr      OF

```



```

fst.WHEN.choice..1st.$choice..$.seq$of$stmts.      END    CASE
';'

        = {printf("  | case$stmt  \n");} ;
opt.iter$spec.:
        = {printf("  | opt.iter$spec.  \n");} ;
        | iter$spec
        = {printf("  | opt.iter$spec.  \n");} ;
loop$stmt: opt.iter$spec.  basic$loop
        = {printf("  | loop$stmt  \n");} ;
opt.id.:
        = {printf("  | opt.id.  \n");}
        | id
        = {printf("  | opt.id.  \n");} ;
basic$loop: LOOP  seq$of$stmts  END  LOOP  opt.id.  ';'
        = {printf("  | basic$loop  \n");} ;
opt.REVERSE.:
        = {printf("  | opt.REVERSE.  \n");}
        | REVERSE
        = {printf("  | opt.REVERSE.  \n");} ;
iter$spec: FOR  loop$para  IN  opt.REVERSE.  disc$range
        = {printf("  | iter$spec  \n");}
        | WHILE  cond
        = {printf("  | iter$spec  \n");} ;
loop$para: id

```



```

        = {printf("    | loop$para  \n");} ;

opt.WHEN.cond.:
        = {printf("    | opt.WHEN.cond.  \n");}

    | WHEN  cond
        = {printf("    | opt.WHEN.cond.  \n");} ;

exit$stmt: EXIT  opt.id.  opt.WHEN.cond.  ';'

        = {printf("    | exit$stmt  \n");} ;

goto$stmt: GOTO  id  ';'

        = {printf("    | goto$stmt  \n");} ;

assert$stmt: ASSERT  cond  ';'

        = {printf("    | assert$stmt  \n");} ;

opt.use$clause.:
        = {printf("    | opt.use$clause.  \n");}

    | use$clause
        = {printf("    | opt.use$clause.  \n");} ;

fst.decl.:
        = {printf("    | fst.decl.  \n");}

    | fst.decl.  decl
        = {printf("    | fst.decl.  \n");} ;

fst.body.:
        = {printf("    | fst.body.  \n");}

    | fst.body.  body
        = {printf("    | fst.body.  \n");} ;

declarative$part: opt.use$clause.  fst.decl.  fst.body.

```



```

        = {printf("    | declarative$part  \n");} ;

body: opt.vis$restriction.  unit$body

        = {printf("    | body  \n");} ;

unit$body: subprog$body

        = {printf("    | unit$body  \n");}

    | mod$spec

        = {printf("    | unit$body  \n");}

    | mod$body

        = {printf("    | unit$body  \n");} ;

opt.formal$part.:

        = {printf("    | opt.formal$part.  \n");}

    | formal$part

        = {printf("    | opt.formal$part.  \n");} ;

opt.RETURN.type$mark.:

        = {printf("    | opt.RETURN.type$mark.  \n");}

    | RETURN  type$mark

        = {printf("    | opt.RETURN.type$mark.  \n");} ;

subprog$spec: subprog$nature  designator  opt.formal$part.

opt.RETURN.type$mark.

        = {printf("    | subprog$spec  \n");} ;

subprog$nature: FUNCTION

        = {printf("    | subprog$nature  \n");}

    | PROCEDURE

        = {printf("    | subprog$nature  \n");} ;

```



```

designator: id

                = {printf("  ! designator  \n");} ;

fst.$.para$decl.:

                = {printf("  ! fst.$.para$decl.  \n");}

                ! fst.$.para$decl. ',' para$decl

                = {printf("  ! fst.$.para$decl.  \n");} ;

formal$part: '(' para$decl fst.$.para$decl. ')'

                = {printf("  ! formal$part  \n");} ;

para$decl: id$list ':' mode type$mark opt.$.expr.

                = {printf("  ! para$decl  \n");} ;

opt.IN.:

                = {printf("  ! opt.IN.  \n");}

                ! IN

                = {printf("  ! opt.IN.  \n");} ;

mode: opt.IN.

                ! OUT

                ! IN OUT ;

subprog$body: subprog$spec IS declarative$part BEGIN

seq$of$stmts END id ';'

                = {printf("  ! subprog$body  \n");} ;

opt.IS.declarative$part.:

                = {printf("  ! opt.IS.declarative$part.

\n");}

                ! IS declarative$part

```



```

        = {printf("      |   opt.IS.declarative$part.
\n"));} ;

mod$spec:  mod$nature      id      opt.IS.declarative$part.      END
opt.id.

        = {printf("      | mod$spec  \n"));} ;

mod$nature: PACKAGE

        = {printf("      | mod$nature  \n"));} ;

mod$body:  mod$nature  BODY  id      IS      declarative$part      END
id  ';'

        = {printf("      | mod$body    \n"));} ;

opt.vis$list.:

        = {printf("      | opt.vis$list.  \n"));}

      | vis$list

        = {printf("      | opt.vis$list.  \n"));} ;

vis$restriction: RESTRICTED  opt.vis$list.

        = {printf("      | vis$restriction  \n"));} ;

fst.$.name.:

        = {printf("      | fst.$.name.  \n"));}

      | fst.$.name.  ','  name

        = {printf("      | fst.$.name.  \n"));} ;

vis$list:  '('  name  fst.$.name.  ')'

        = {printf("      | vis$list  \n"));} ;

use$clause:  USE  name  fst.$.name.

        = {printf("      | use$clause  \n"));} ;

```


%%

#include "lex.yy.c"

APPENDIX C

Parse Table

This Appendix illustrates the output of the YACC program used to manually interpret YACC's parse tables. This file is produced using the "- v" option of YACC.

state 0

```
$accept : ← compilation$unit $end
opt.vis$restriction. : ←      (2)
RESTRICTED shift 4
. reduce 2
compilation$unit goto 1
opt.vis$restriction. goto 2
vis$restriction goto 3
```

state 1

```
$accept : compilation$unit←$end
$end accept
. error
```

state 2

```
compilation$unit :
opt.vis$restriction.←opt.SEPARATE. unit$body
opt.SEPARATE. : ←      (4)
SEPARATE shift 6
```



```

    . reduce 4
    opt.SEPARATE. goto 5

state 3
    opt.vis$restriction. : vis$restriction← (3)
    . reduce 3

state 4
    vis$restriction : RESTRICTED←opt.vis$list.
    opt.vis$list. : ← (198)
    ( shift 9
    . reduce 198
    opt.vis$list. goto 7
    vis$list goto 8

state 5
    comoilation$unit : opt.vis$restriction.
    opt.SEPARATE.←unit$body

    FUNCTION shift 18
    PACKAGE shift 17
    PROCEDURE shift 19
    . error
    unit$body goto 10
    subprog$body goto 11
    mod$spec goto 12
    mod$body goto 13
    subprog$spec goto 14

```



```

        subprog$nature goto 16
        mod$nature goto 15

state 6
        opt.SEPARATE. : SEPARATE+      (5)
        . reduce 5

state 7
        vis$restriction : RESTRICTED opt.vis$list.+      (200)
        . reduce 200

state 8
        opt.vis$list. : vis$list+      (199)
        . reduce 199

state 9
        vis$list : (←name fst.$.name. )
        id shift 21
        . error
        name goto 20
        subprog$array$var goto 22
        selected$comp goto 23
        predefined$attri goto 24

state 10
        compilation$unit : opt.vis$restriction.
        opt.SEPARATE. unit$body+      (1)
        . reduce 1

state 11

```



```

        unit$body : subprog$body+      (172)
        . reduce 172

state 12
        unit$body : mod$spec+      (173)
        . reduce 173

state 13
        unit$body : mod$body+      (174)
        . reduce 174

state 14
        subprog$body : subprog$spec+IS declarative$part BE-
GIN seq$of$stmts END id ;
        IS shift 25
        . error

state 15
        mod$spec : mod$nature+id opt.IS.declarative$part.
END opt.id.
        mod$body : mod$nature+BODY id IS declarative$part
END id ;
        BODY shift 27
        id shift 26
        . error

state 16
        subprog$spec : subprog$nature+designator
opt.formal$part. opt.RETURN.type$mark.

```



```

    id shift 29
    . error
    designator goto 28

state 17
    mod$nature : PACKAGE← (196)
    . reduce 196

state 18
    subprog$nature : FUNCTION← (180)
    . reduce 180

state 19
    subprog$nature : PROCEDURE← (181)
    . reduce 181

state 20
    selected$comp : name←. id
    predefined$attri : name←' id
    subprog$array$var : name←( expr fst.$expr. )
    vis$list : ( name←fst.$name. )
    fst.$name. : + (201)
    ( shift 32
    . shift 30
    ' shift 31
    . reduce 201
    fst.$name. goto 33

state 21

```



```

        name : id←      (58)
        . reduce 58

state 22
        name : subprog$array$var←      (59)
        . reduce 59

state 23
        name : selected$comp←      (60)
        . reduce 60

state 24
        name : predefined$attri←      (61)
        . reduce 61

state 25
        suborog$body : subprog$spec IS←declarative$part BE-
GIN seq$of$stmts END id ;
        opt.use$clause. : ←      (164)
        USE shift 37
        . reduce 164
        opt.use$clause. goto 35
        use$clause goto 36
        declarative$part goto 34

state 26
        mod$spec : mod$nature id←opt.IS.declarative$part.
END opt.id.
        opt.IS.declarative$part. : ←      (193)

```



```

    IS shift 39
    . reduce 193
    opt.IS.declarative$part. goto 38
state 27
    mod$body : mod$nature BODY←id IS declarative$part
END id ;
    id shift 40
    . error
state 28
    subprog$spec          :          subprog$nature
designator←opt.formal$part. opt.RETURN.type$mark.
    opt.formal$part. : ← (175)
    ( shift 43
    . reduce 175
    opt.formal$part. goto 41
    formal$part goto 42
state 29
    designator : id← (182)
    . reduce 182
state 30
    selected$comp : name .←id
    id shift 44
    . error
state 31

```



```

predefined$attri : name '←id
id shift 45
. error

```

state 32

```

subprog$array$var : name (←expr fst.$expr. )
opt.unary$op. : ← (81)
NOT shift 53
+ shift 51
- shift 52
. reduce 81
expr goto 46
simple$expr goto 48
rel goto 47
opt.unary$op. goto 49
unary$op goto 50

```

state 33

```

fst.$name. : fst.$name.←, name
vis$list : ( name fst.$name.←)
) shift 55
, shift 54
. error

```

state 34

```

subprog$body      :      subprog$spec      IS
declarative$part←BEGIN seq$of$stmts END id ;

```



```

        BEGIN  shift 56

        .  error

state 35

        declarative$part      :      opt.use$clause.←fst.decl.
fst.body.

        fst.decl. : ←      (166)

        .  reduce 166

        fst.decl.  goto 57

state 36

        opt.use$clause. :  use$clause←      (165)

        .  reduce 165

state 37

        use$clause :  USE←name fst.$name.

        id  shift 21

        .  error

        name  goto 58

        subprog$array$var  goto 22

        selected$comp  goto 23

        predefined$attri  goto 24

state 38

        mod$spec      :      mod$nature      id

        opt.IS.declarative$part.←END opt.id.

        END  shift 59

        .  error

```


state 39

opt.IS.declarative\$part. : IS←declarative\$part

opt.use\$clause. : ← (164)

USE shift 37

. reduce 164

opt.use\$clause. goto 35

use\$clause goto 36

declarative\$part goto 60

state 40

mod\$body : mod\$nature BODY id←IS declarative\$part

END id ;

IS shift 61

. error

state 41

subprog\$spec : subprog\$nature designator

opt.formal\$part.←opt.RETURN.type\$mark.

opt.RETURN.type\$mark. : ← (177)

RETURN shift 63

. reduce 177

opt.RETURN.type\$mark. goto 62

state 42

opt.formal\$part. : formal\$part← (176)

. reduce 176

state 43


```
formal$part : (←para$decl fst.$para$decl. )
```

```
id shift 66
```

```
. error
```

```
id$list goto 65
```

```
para$decl goto 64
```

```
state 44
```

```
selected$comp : name . id← (62)
```

```
. reduce 62
```

```
state 45
```

```
predefined$attri : name ' id← (63)
```

```
. reduce 63
```

```
state 46
```

```
subprog$array$var : name ( expr←fst.$expr. )
```

```
expr : expr←log$op rel
```

```
fst.$expr. : ← (67)
```

```
AND shift 69
```

```
OR shift 70
```

```
XOR shift 71
```

```
. reduce 67
```

```
fst.$expr. goto 67
```

```
log$op goto 68
```

```
state 47
```

```
expr : rel← (80)
```

```
. reduce 80
```


state 48

```
rel : simple$expr←opt.relal$op.simple$expr.  
rel : simple$expr←opt.NOT. IN range  
rel : simple$expr←opt.NOT. IN type$mark opt.constr.  
opt.relal$op.simple$expr. : ← (70)  
opt.NOT. : ← (72)  
IN reduce 72  
NOT shift 75  
/? shift 77  
<= shift 79  
>= shift 81  
= shift 76  
< shift 78  
> shift 80  
. reduce 70  
opt.relal$op.simple$expr. goto 72  
relal$op goto 74  
opt.NOT. goto 73
```

state 49

```
simple$expr : opt.unary$op.←term fst.adding$op.term.  
id shift 21  
num shift 89  
char$str shift 90  
( shift 91
```



```

. error
name goto 87
lit goto 85
aggr goto 86
pri goto 84
subprog$array$var goto 22
selected$comp goto 23
predefined$attri goto 24
var goto 88
term goto 82
fac goto 83

state 50
    opt.unary$op. : unary$op+ (82)
    . reduce 82

state 51
    unary$op : ++ (106)
    . reduce 106

state 52
    unary$op : -+ (107)
    . reduce 107

state 53
    unary$op : NOT+ (108)
    . reduce 108

state 54

```



```

    fst.$.name. :  fst.$.name. ,←name
    id  shift 21
    .  error
    name  goto 92
    subprog$array$var  goto 22
    selected$comp  goto 23
    predefined$attri  goto 24
state 55
    vis$list :  ( name fst.$.name. )←      (203)
    .  reduce 203
state 56
    subrog$body :  subrog$spec  IS  declarative$part
BEGIN←seq$of$stmts END id ;
    fst.stmt. : ←      (112)
    .  reduce 112
    fst.stmt.  goto 94
    seq$of$stmts  goto 93
state 57
    fst.decl. :  fst.decl.←decl
    declarative$part      :      opt.use$clause.
fst.decl.←fst.body.
    fst.body. : ←      (168)
    error  shift 99
    TYPE  shift 101

```



```

id shift 66
. reduce 168
decl goto 95
obj$decl goto 97
type$decl goto 98
id$list goto 100
fst.body. goto 96

```

state 58

```

selected$comp : name←. id
predefined$attri : name←' id
subprog$array$var : name←( expr fst.$.expr. )
use$clause : USE name←fst.$.name.
fst.$.name. : ← (201)
( shift 32
. shift 30
' shift 31
. reduce 201
fst.$.name. goto 102

```

state 59

```

mod$spec : mod$nature id opt.IS.declarative$part.
END←opt.id.
opt.id. : ← (151)
id shift 104
. reduce 151

```



```

        opt.id. goto 103

state 60
        opt.IS.declarative$part. : IS declarative$part←
(194)
        . reduce 194

state 61
        mod$body : mod$nature BODY id IS←declarative$part
END id ;

        opt.use$clause. : ← (164)
        USE shift 37
        . reduce 164
        opt.use$clause. goto 35
        use$clause goto 36
        declarative$part goto 105

state 62
        subprog$spec : subprog$nature designator
opt.formal$part. opt.RETURN.type$mark.← (179)
        . reduce 179

state 63
        opt.RETURN.type$mark. : RETURN←type$mark
        id shift 21
        . error
        type$mark goto 106
        name goto 107

```



```

subprog$array$var goto 22
selected$comp goto 23
predefined$attri goto 24

state 64
    formal$part : ( para$decl←fst.$.para$decl. )
    fst.$.para$decl. : ← (183)
    . reduce 183
    fst.$.para$decl. goto 108

state 65
    para$decl : id$list←: mode type$mark opt.$.expr.
    : shift 109
    . error

state 66
    id$list : id←fst$.id.
    fst$.id. : ← (14)
    . reduce 14
    fst$.id. goto 110

state 67
    fst$.expr. : fst$.expr.←, expr
    subprog$array$var : name ( expr fst$.expr.←)
    ) shift 112
    , shift 111
    . error

state 68

```



```

expr : expr log$op+rel
opt.unary$op. : +      (81)
NOT shift 53
+ shift 51
- shift 52
. reduce 81
simple$expr goto 48
rel goto 113
opt.unary$op. goto 49
unary$op goto 50

```

state 69

```

log$op : AND+      (94)
. reduce 94

```

state 70

```

log$op : OR+       (95)
. reduce 95

```

state 71

```

log$op : XOR+      (96)
. reduce 96

```

state 72

```

rel : simple$expr opt.relal$op.simple$expr.+      (76)
. reduce 76

```

state 73

```

rel : simple$expr opt.NOT.+IN range

```



```

rel : simple$expr opt.NOT.←IN type$mark opt.constr.
IN shift 114
. error

```

state 74

```

opt.relal$op.simple$expr. : relal$op←simple$expr
opt.unary$op. : ← (81)
NOT shift 53
+ shift 51
- shift 52
. reduce 81
simple$expr goto 115
opt.unary$op. goto 49
unary$op goto 50

```

state 75

```

opt.NOT. : NOT← (73)
. reduce 73

```

state 76

```

relal$op : =← (97)
. reduce 97

```

state 77

```

relal$op : /=← (98)
. reduce 98

```

state 78

```

relal$op : <← (99)

```



```

        . reduce 99

state 79

    relal$op : <=←    (100)

    . reduce 100

state 80

    relal$op : >←    (101)

    . reduce 101

state 81

    relal$op : >=←    (102)

    . reduce 102

state 82

    simple$expr : opt.unary$op. term←fst.adding$op.term.
    fst.adding$op.term. : ←    (83)

    . reduce 83

    fst.adding$op.term. goto 116

state 83

    term : fac←fst.mult$op.fac.
    fst.mult$op.fac. : ←    (86)

    . reduce 86

    fst.mult$op.fac. goto 117

state 84

    fac : pri←    (89)

    . reduce 89

state 85

```



```

    pri : lit+      (90)
    . reduce 90

state 86
    pri : aggr+     (91)
    . reduce 91

state 87
    selected$comp : name+. id
    predefined$attri : name+' id
    var : name+. ALL
    subprog$array$var : name+( expr fst.$.expr. )
    pri : name+     (92)
    ( shift 32
    . shift 118
    ' shift 31
    . reduce 92

state 88
    pri : var+      (93)
    . reduce 93

state 89
    lit : num+      (64)
    . reduce 64

state 90
    lit : char$str+ (65)
    . reduce 65

```


state 91

aggr : (←comp\$assoc fst.\$comp\$assoc.)

OTHERS shift 123

id shift 21

num shift 89

char\$str shift 90

(shift 91

. error

type\$mark goto 125

name goto 124

lit goto 85

disc\$range goto 122

comp\$assoc goto 119

aggr goto 86

restr\$choice goto 120

pri goto 121

subprog\$array\$var goto 22

selected\$comp goto 23

predefined\$attri goto 24

var goto 88

state 92

selected\$comp : name←. id

predefined\$attri : name←' id

subprog\$array\$var : name←(expr fst.\$expr.)

fst.\$.name. : fst.\$.name. , name← (202)

(shift 32

. shift 30

' shift 31

. reduce 202

state 93

subprog\$body : subprog\$spec IS declarative\$part BE-
GIN seq\$of\$stmts←END id ;

END shift 126

. error

state 94

fst.stmt. : fst.stmt.←stmt

seq\$of\$stmts : fst.stmt.← (114)

opt.iter\$spec. : ← (148)

error shift 131

ASSERT shift 147

CASE shift 149

EXIT shift 144

FOR shift 152

GOTO shift 146

IF shift 148

LOOP reduce 148

NULL shift 138

RETURN shift 145


```
WHILE shift 153  
id shift 21  
<< shift 130  
. reduce 114  
name goto 143  
subprog$array$var goto 22  
selected$comp goto 23  
predefined$attri goto 24  
var goto 142  
stmt goto 127  
simple$stmt goto 128  
compound$stmt goto 129  
assignment$stmt goto 132  
subprog$call$stmt goto 133  
exit$stmt goto 134  
return$stmt goto 135  
goto$stmt goto 136  
assert$stmt goto 137  
if$stmt goto 139  
case$stmt goto 140  
loop$stmt goto 141  
opt.iter$spec. goto 150  
iter$spec goto 151
```

state 95


```

    fst.decl. : fst.decl. decl←      (167)
    . reduce 167

state 96

    fst.body. : fst.body.←body
    declarative$part : opt.use$clause.    fst.decl.
fst.body.←      (170)

    opt.vis$restriction. : ←      (2)
BEGIN reduce 170
END reduce 170
RESTRICTED shift 4
    . reduce 2
    opt.vis$restriction. goto 155
    vis$restriction goto 3
    body goto 154

state 97

    decl : obj$decl←      (6)
    . reduce 6

state 98

    decl : type$decl←      (7)
    . reduce 7

state 99

    decl : error←;
    ; shift 156
    . error

```


state 100

obj\$decl : id\$list←: opt.CONSTANT. type opt\$.expr.

;

: shift 157

. error

state 101

type\$decl : TYPE←id IS type\$defn ;

id shift 158

. error

state 102

fst\$.name. : fst\$.name.←, name

use\$clause : USE name fst\$.name.← (204)

, shift 54

. reduce 204

state 103

mod\$spec : mod\$nature id opt.IS.declarative\$part.

END opt.id.← (195)

. reduce 195

state 104

opt.id. : id← (152)

. reduce 152

state 105

mod\$body : mod\$nature BODY id IS

declarative\$part←END id ;


```

END shift 159

. error

state 106

opt.RETURN.type$mark. : RETURN type$mark+ (178)

. reduce 178

state 107

type$mark : name+ (22)
selected$comp : name+. id
predefined$attri : name+' id
subprog$array$var : name+( expr fst.$expr. )
( shift 32
. shift 30
' shift 31
. reduce 22

state 108

fst.$para$decl. : fst.$para$decl.+ , para$decl
formal$part : ( para$decl fst.$para$decl.+ )
) shift 161
, shift 160
. error

state 109

para$decl : id$list :+mode type$mark opt.$expr.
opt.IN. : + (187)
IN shift 165

```


OUT shift 164

. reduce 187

mode goto 162

opt.IN. goto 163

state 110

fst\$.id. : fst\$.id.←, id

id\$list : id fst\$.id.← (16)

, shift 166

. reduce 16

state 111

fst\$.expr. : fst\$.expr. ,←expr

opt.unary\$op. : ← (81)

NOT shift 53

+ shift 51

- shift 52

. reduce 81

expr goto 167

simple\$expr goto 48

rel goto 47

opt.unary\$op. goto 49

unary\$op goto 50

state 112

subprog\$array\$var : name (expr fst\$.expr.)←

(69)


```

        . reduce 69

state 113
    expr : expr log$op rel+      (79)
        . reduce 79

state 114
    rel : simple$expr opt.NOT. IN+range
    rel : simple$expr opt.NOT. IN+type$mark opt.constr.
    id shift 171
    num shift 89
    char$str shift 90
    . error
    type$mark goto 169
    name goto 107
    range goto 168
    range$id goto 170
    lit goto 172
    subprog$array$var goto 22
    selected$comp goto 23
    predefined$attri goto 24

state 115
    opt.relal$op.simple$expr. : relal$op simple$expr+
(71)
        . reduce 71

state 116

```



```

        fst.adding$op.term. :   fst.adding$op.term.←adding$op
term

        simple$expr      :           opt.unary$op.           term
fst.adding$op.term.←      (85)
    &  shift 176
    +  shift 174
    -  shift 175
    .  reduce 85
    adding$op  goto 173

state 117

    fst.mult$op.fac. :   fst.mult$op.fac.←mult$op fac
    term :   fac fst.mult$op.fac.←      (88)
    MOD  shift 180
    *  shift 178
    /  shkfu 179
    .  reduce 88
    mult$op  goto 177

state 118

    selected$comp :   name .←id
    var :   name .←ALL
    ALL  shift 181
    id  shift 44
    .  error

state 119

```


aggr : (comp\$assoc←fst.\$.comp\$assoc.)

fst.\$.comp\$assoc. : ← (41)

. reduce 41

fst.\$.comp\$assoc. goto 182

state 120

comp\$assoc : restr\$choice←fst.\$.choice. => expr

fst.\$.choice. : ← (44)

. reduce 44

fst.\$.choice. goto 183

state 121

comp\$assoc : pri←opt..1st.\$.choice..\$.expr.

opt..1st.\$.choice..\$.expr. : ← (46)

fst.\$.choice. : ← (44)

) reduce 46

, reduce 46

. reduce 44

fst.\$.choice. goto 185

opt..1st.\$.choice..\$.expr. goto 184

state 122

restr\$choice : disc\$range← (50)

. reduce 50

state 123

restr\$choice : OTHERS← (5q)

. reduce 51

state 124

```
type$mark : name← (22)
selected$comp : name←. id
predefined$attri : name←' id
var : name←. ALL
subprog$array$var : name←( expr fst.$expr. )
pri : name← (92)
RANGE reduce 22
( shift 32
. shift 118
' shift 31
. reduce 92
```

state 125

```
disc$range : type$mark←RANGE range
RANGE shift 186
. error
```

state 126

```
subprog$body : subprog$spec IS declarative$part BE-
GIN seq$of$stmts END←id ;
id shift 187
. error
```

state 127

```
fst.stmt. : fst.stmt. stmt← (113)
. reduce 113
```


state 128

stmt : simple\$stmt← (115)

. reduce 115

state 129

stmt : compound\$stmt← (116)

. reduce 116

state 130

stmt : <<id >> stmt

id shift 188

. error

state 131

stmt : error;

; shift 189

. error

state 132

simple\$stmt : assignment\$stmt← (119)

. reduce 119

state 133

simple\$stmt : subprog\$call\$stmt← (120)

. reduce 120

state 134

simple\$stmt : exit\$stmt← (121)

. reduce 121

state 135


```

        simple$stmt : return$stmt←      (122)
        . reduce 122

state 136
        simple$stmt : goto$stmt←      (123)
        . reduce 123

state 137
        simple$stmt : assert$stmt←     (124)
        . reduce 124

state 138
        simple$stmt : NULL←;
        ; shift 190
        . error

state 139
        compound$stmt : if$stmt←       (126)
        . reduce 126

state 140
        compound$stmt : case$stmt←     (127)
        . reduce 127

state 141
        compound$stmt : loop$stmt←     (128)
        . reduce 128

state 142
        assignment$stmt : var←:= expr ;
        :? shift 191

```


. error

state 143

```
selected$comp : name←. id
predefined$attri : name←' id
var : name←. ALL
subprog$array$var : name←( expr fst.$expr. )
assignment$stmt : name←:= expr ;
subprog$call$stmt : name←;
:= shift 192
; shift 193
( shift 32
. shift 118
' shift 31
. error
```

state 144

```
exit$stmt : EXIT←opt.id. opt.WHEN.cond. ;
opt.id. : ← (151)
id shift 104
. reduce 151
opt.id. goto 194
```

state 145

```
return$stmt : RETURN←opt.expr. ;
opt.expr. : ← (132)
opt.unary$op. : ← (81)
```



```

NOT  shift 53
;   reduce 132
+   shift 51
-   shift 52
.   reduce 81
expr goto 196
simple$expr goto 48
rel  goto 47
opt.unary$op. goto 49
unary$op goto 50
opt.expr. goto 195

```

state 146

```

goto$stmt : GOTO+id ;
id  shift 197
.   error

```

state 147

```

assert$stmt : ASSERT+cond ;
opt.unary$op. : + (81)
NOT  shift 53
+   shift 51
-   shift 52
.   reduce 81
expr goto 199
simple$expr goto 48

```



```

rel goto 47
opt.unary$op. goto 49
unary$op goto 50
cond goto 198

```

state 148

```

if$stmt      :      IF←cond      THEN      seq$of$stmts
fst.ELIF.cond.THEN.seq$of$stmts. opt.ELSE.seq$of$stmts. END
IF ;

```

```

opt.unary$op. : ←      (81)
NOT shift 53
+ shift 51
- shift 52
. reduce 81
expr goto 199
simple$expr goto 48
rel goto 47
opt.unary$op. goto 49
unary$op goto 50
cond goto 200

```

state 149

```

case$stmt      :      CASE←expr      OF
fst.WHEN.choice..1st.$choice..$.seq$of$stmts. END CASE
opt.unary$op. : ←      (81)
NOT shift 53

```



```

+ shift 51
- shift 52
. reduce 81
expr goto 201
simple$expr goto 48
rel goto 47
opt.unary$op. goto 49
unary$op goto 50

state 150
    loop$stmt : opt.iter$spec.←basic$loop
    LOOP shift 203
    . error
    basic$loop goto 202

state 151
    opt.iter$spec. : iter$spec← (149)
    . reduce 149

state 152
    iter$spec : FOR←loop$para IN opt.REVERSE. disc$range
    id shift 205
    . error
    loop$para goto 204

state 153
    iter$spec : WHILE←cond
    opt.unary$op. : ← (81)

```



```

NOT shift 53
+ shift 51
- shift 52
. reduce 81
expr goto 199
simple$expr goto 48
rel goto 47
opt.unary$op. goto 49
unary$op goto 50
cond goto 206

```

state 154

```

fst.body. : fst.body. body← (169)
. reduce 169

```

state 155

```

body : opt.vis$restriction.←unit$body
FUNCTION shift 18
PACKAGE shift 17
PROCEDURE shift 19
. error
unit$body goto 207
subprog$body goto 11
mod$spec goto 12
mod$body goto 13
subprog$spec goto 14

```



```

    subprog$nature goto 16
    mod$nature goto 15
state 156
    decl : error ;← (8)
    . reduce 8
state 157
    obj$decl : id$list :←opt.CONSTANT. type opt.$expr. ;
    opt.CONSTANT. : ← (9)
    CONSTANT shift 209
    . reduce 9
    opt.CONSTANT. goto 208
state 158
    type$decl : TYPE id←IS type$defn ;
    IS shift 210
    . error
state 159
    mod$body : mod$nature BODY id IS declarative$part
END←id ;
    id shift 211
    . error
state 160
    fst.$para$decl. : fst.$para$decl. ,←para$decl
    id shift 66
    . error

```



```

        id$list goto 65
        para$decl goto 212
state 161
        formal$part : ( para$decl fst.$para$decl. )←
(185)
        . reduce 185
state 162
        para$decl : id$list : mode←type$mark opt.$expr.
        id shift 21
        . error
        type$mark goto 213
        name goto 107
        subprog$array$var goto 22
        selected$comp goto 23
        predefined$attri goto 24
state 163
        mode : opt.IN.← (189)
        . reduce 189
state 164
        mode : OUT← (190)
        . reduce 190
state 165
        opt.IN. : IN← (188)
        mode : IN←OUT

```



```

    OUT  shift 214
    .  reduce 188

state 166
    fst$.id. :  fst$.id. ,+id
    id  shift 215
    .  error

state 167
    fst$.expr. :  fst$.expr. , expr+      (68)
    expr :  expr+log$op rel
    AND  shift 69
    OR  shift 70
    XOR  shift 71
    .  reduce 68
    log$op  goto 68

state 168
    rel :  simple$expr opt.NOT. IN range+      (77)
    .  reduce 77

state 169
    rel :  simple$expr opt.NOT. IN type$mark+opt.constr.
    opt.constr. :  +      (74)
    RANGE  shift 219
    .  reduce 74
    constr  goto 217
    range$constr  goto 218

```



```

    opt.constr. goto 216

state 170
    range : range$id←.. siople$expr
    .. shift 220
    . error

state 171
    range$id : id← (27)
    name : id← (58)
    .. reduce 27
    => reduce 27
    ! reduce 27
    . reduce 58

state 172
    range$id : lit← (28)
    . reduce 28

state 173
    fst.adding$op.tero. : fst.adding$op.term.
adding$op←term
    id shift 21
    num shift 89
    char$str shift 90
    ( shift 91
    . error
    name goto 87

```



```

lit goto 85
aggr goto 86
pri goto 84
subprog$array$var goto 22
selected$comp goto 23
predefined$attri goto 24
var goto 88
term goto 221
fac goto 83

```

state 174

```

adding$op : +← (103)
. reduce 103

```

state 175

```

adding$op : -← (104)
. reduce 104

```

state 176

```

adding$op : &← (105)
. reduce 105

```

state 177

```

fst.mult$op.fac. : fst.mult$op.fac. mult$op+fac
id shift 21
num shift 89
char$str shift 90
( shift 91

```



```

. error
name goto 87
lit goto 85
aggr goto 86
pri goto 84
subprog$array$var goto 22
selected$comp goto 23
predefined$attri goto 24
var goto 88
fac goto 222

state 178
    mult$op : *← (109)
    . reduce 109

state 179
    mult$op : /← (110)
    . reduce 110

state 180
    mult$op : MOD← (111)
    . reduce 111

state 181
    var : name . ALL← (66)
    . reduce 66

state 182
    fst$.comp$assoc. : fst$.comp$assoc.←, comp$assoc

```



```
aggr : ( como$assoc fst.$comp$assoc.+)
) shift 224
, shift 223
. error
```

state 183

```
fst.$choice. : fst.$choice.+! choice
comp$assoc : restr$choice fst.$choice.+=> expr
=> shift 226
! shift 225
. error
```

state 184

```
comp$assoc : pri opt..1st.$choice..$.expr.+ (49)
. reduce 49
```

state 185

```
fst.$choice. : fst.$choice.+! choice
opt..1st.$choice..$.expr. : fst.$choice.+?> expr
=> shift 227
! shift 225
. error
```

state 186

```
disc$range : type$mark RANGE+range
id shift 229
num shift 89
char$str shift 90
```



```

    . error
    range goto 228
    range$id goto 170
    lit goto 172

state 187
    subprog$body : subprog$spec IS declarative$part BE-
GIN seq$of$stmts END id←;
    ; shift 230
    . error

state 188
    stmt : << id<>> stmt
    >> shift 231
    . error

state 189
    stmt : error ;← (118)
    . reduce 118

state 190
    simple$stmt : NULL ;← (125)
    . reduce 125

state 191
    assignment$stmt : var :=←expr ;
    opt.unary$op. : ← (81)
    NOT shift 53
    + shift 51

```



```

- shift 52
. reduce 81
expr goto 232
simple$expr goto 48
rel goto 47
opt.unary$op. goto 49
unary$op goto 50

```

state 192

```

assignment$stmt : name :=←expr ;
opt.unary$op. : ← (81)
NOT shift 53
+ shift 51
- shift 52
. reduce 81
expr goto 233
simple$expr goto 48
rel goto 47
opt.unary$op. goto 49
unary$op goto 50

```

state 193

```

subprog$call$stmt : name ;← (131)
. reduce 131

```

state 194

```

exit$stmt : EXIT opt.id.←opt.WHEN.cond. ;

```


opt.WHEN.cond. : + (159)

WHEN shift 235

. reduce 159

opt.WHEN.cond. goto 234

state 195

return\$stmt : RETURN opt.expr.;

; shift 236

. error

state 196

expr : expr+log\$op rel

opt.expr. : expr+ (133)

AND shift 69

OR shift 70

XOR shift 71

. reduce 133

log\$op goto 68

state 197

goto\$stmt : GOTO id.;

; shift 237

. error

state 198

assert\$stmt : ASSERT cond.;

; shift 238

. error

state 199

```
    expr : expr+log$op rel
    cond : expr+fst.cond$ext.
    fst.cond$ext. : +      (140)
    AND  shift 69
    OR   shift 70
    XOR  shift 71
    .    reduce 140
    log$op goto 68
    fst.cond$ext. goto 239
```

state 200

```
    if$stmt      :      IF      cond+THEN      seq$of$stmts
fst.ELIF.cond.THEN.seq$of$stots. opt.ELSE.seq$of$stmts. END
IF ;

    THEN  shift 240
    .    error
```

state 201

```
    expr : expr+log$op rel
    case$stmt      :      CASE      expr+OF
fst.WHEN.choice..1st.$choice..$.seq$of$stmts. END CASE

    AND  shift 69
    OF   shift 241
    OR   shift 70
    XOR  shift 71
```



```

    . error

    log$op goto 68

state 202

    loop$stmt : opt.iter$spec. basic$loop+      (150)

    . reduce 150

state 203

    basic$loop : LOOP+seq$of$stmts END LOOP opt.id. ;

    fst stmt. : +      (112)

    . reduce 112

    fst stmt. goto 94

    seq$of$stmts goto 242

state 204

    iter$spec : FOR loop$para+IN opt.REVERSE. disc$range

    IN shift 243

    . error

state 205

    loop$para : id+      (158)

    . reduce 158

state 206

    iter$spec : WHILE cond+      (157)

    . reduce 157

state 207

    body : opt.vis$restriction. unit$body+      (171)

    . reduce 171

```


state 208

```
obj$decl : id$list : opt.CONSTANT.←type opt.$expr. ;  
id shift 21  
. error  
type goto 244  
type$mark goto 245  
name goto 107  
subprog$array$var goto 22  
selected$comp goto 23  
predefined$attri goto 24
```

state 209

```
opt.CONSTANT. : CONSTANT← (10)  
. reduce 10
```

state 210

```
type$decl : TYPE id IS←type$defn ;  
ARRAY shift 253  
RANGE shift 219  
RECORD shift 254  
( shift 251  
. error  
type$defn goto 246  
enum$type$defn goto 247  
int$type$defn goto 248  
array$type$defn goto 249
```



```

        record$type$defn goto 250
        range$constr goto 252
state 211
        mod$body : mod$nature BODY id IS declarative$part
END id+;
        ; shift 255
        . error
state 212
        fst$.para$decl. : fst$.para$decl. , para$decl+
(184)
        . reduce 184
state 213
        para$decl : id$list : mode type$mark+opt$.expr.
        opt$.expr. : + (11)
        := shift 257
        . reduce 11
        opt$.expr. goto 256
state 214
        mode : IN OUT+ (191)
        . reduce 191
state 215
        fst$.id. : fst$.id. , id+ (15)
        . reduce 15
state 216

```



```

        rel      :      simple$expr      opt.NOT.      IN      type$mark
opt.constr.+      (78)
        .      reduce 78
state 217
        opt.constr. :      constr+      (75)
        .      reduce 75
state 218
        constr :      range$constr+      (23)
        .      reduce 23
state 219
        range$constr :      RANGE+range
        id      shift 229
        num      shift 89
        char$str      shift 90
        .      error
        range      goto 258
        range$id      goto 170
        lit      goto 172
state 220
        range :      range$id ..+simple$expr
        opt.unary$op. :      +      (81)
        NOT      shift 53
        +      shift 51
        -      shift 52

```



```

    . reduce 81
    simple$expr goto 259
    opt.unary$op. goto 49
    unary$op goto 50

state 221
    fst.adding$op.term. : fst.adding$op.term. adding$op
term← (84)
    . reduce 84

state 222
    fst.mult$op.fac. : fst.mult$op.fac. mult$op fac←
(87)
    . reduce 87

state 223
    fst.$comp$assoc. : fst.$comp$assoc. ,+comp$assoc
OTHERS shift 123
    id shift 21
    num shift 89
    char$str shift 90
    ( shift 91
    . error
    type$mark goto 125
    name goto 124
    lit goto 85
    disc$range goto 122

```



```
comp$assoc goto 260
aggr goto 86
restr$choice goto 120
pri goto 121
subprog$array$var goto 22
selected$comp goto 23
predefined$attri goto 24
var goto 88
```

state 224

```
aggr : ( comp$assoc fst.$.comp$assoc. )← (43)
. reduce 43
```

state 225

```
fst.$.choice. : fst.$.choice. !←choice
OTHERS shift 123
id shift 171
num shift 89
char$str shift 90
. error
type$mark goto 125
name goto 107
range$id goto 262
lit goto 172
disc$range goto 122
choice goto 261
```



```

restr$choice goto 263
subprog$array$var goto 22
selected$comp goto 23
predefined$attri goto 24

```

state 226

```

comp$assoc : restr$choice fst.$.choice. =>←expr
opt.unary$op. : ← (81)
NOT shift 53
+ shift 51
- shift 52
. reduce 81
expr goto 264
simple$expr goto 48
rel goto 47
opt.unary$op. goto 49
unary$op goto 50

```

state 227

```

opt..1st.$.choice..$.expr. : fst.$.choice. =>←expr
opt.unary$op. : ← (81)
NOT shift 53
+ shift 51
- shift 52
. reduce 81
expr goto 265

```



```
simple$expr goto 48
rel goto 47
opt.unary$op. goto 49
unary$op goto 50
```

state 228

```
disc$range : type$mark RANGE range← (40)
. reduce 40
```

state 229

```
range$id : id← (27)
. reduce 27
```

state 230

```
subprog$body : subprog$spec IS declarative$part BE-
GIN seq$of$stmts END id ;← (192)
. reduce 192
```

state 231

```
stmt : << id >>←stmt
opt.iter$spec. : ← (148)
error shift 131
ASSERT shift 147
CASE shift 149
EXIT shift 144
FOR shift 152
GOTO shift 146
IF shift 148
```


NULL shift 138
RETURN shift 145
WHILE shift 153
id shift 21
<< shift 130
. reduce 148
name goto 143
subprog\$array\$var goto 22
selected\$comp goto 23
predefined\$attri goto 24
var goto 142
stmt goto 266
simple\$stmt goto 128
compound\$stmt goto 129
assignment\$stmt goto 132
subprog\$call\$stmt goto 133
exit\$stmt goto 134
return\$stmt goto 135
goto\$stmt goto 136
assert\$stmt goto 137
if\$stmt goto 139
case\$stmt goto 140
loop\$stmt goto 141
opt.iter\$spec. goto 150

iter\$spec goto 151

state 232

expr : expr+log\$op rel

assignment\$stmt : var := expr+;

AND shift 69

OR shift 70

XOR shift 71

; shift 267

. error

log\$op goto 68

state 233

expr : expr+log\$op rel

assignment\$stmt : name := expr+;

AND shift 69

OR shift 70

XOR shift 71

; shift 268

. error

log\$op goto 68

state 234

exit\$stmt : EXIT opt.id. opt.WHEN.cond.+;

; shift 269

. error

state 235

opt.WHEN.cond. : WHEN←cond

opt.unary\$op. : ← (81)

NOT shift 53

+ shift 51

- shift 52

. reduce 81

expr goto 199

simple\$expr goto 48

rel goto 47

opt.unary\$op. goto 49

unary\$op goto 50

cond goto 270

state 236

return\$stmt : RETURN opt.expr. ;← (134)

. reduce 134

state 237

goto\$stmt : GOTO id ;← (162)

. reduce 162

state 238

assert\$stmt : ASSERT cond ;← (163)

. reduce 163

state 239

fst.cond\$ext. : fst.cond\$ext.←cond\$ext

cond : expr fst.cond\$ext.← (142)


```

AND.THEN  shift 272
OR.ELSE  shift 273
.  reduce 142
cond$ext  goto 271

state 240
    if$stmt      :      IF      cond      THEN←seq$of$stmts
fst.ELSIF.cond.THEN.seq$of$stmts. opt.ELSE.seq$of$stmts. END
IF ;

    fst.stmt. : ←      (112)
.  reduce 112
    fst.stmt.  goto 94
    seq$of$stmts  goto 274

state 241
    case$stmt      :      CASE      expr
OF←fst.WHEN.choice..1st.$choice..$.seq$of$stmts. END CASE
    fst.WHEN.choice..1st.$choice..$.seq$of$stmts. : ←
(145)
.  reduce 145
    fst.WHEN.choice..1st.$choice..$.seq$of$stmts.  goto
275

state 242
    basic$loop :  LOOP seq$of$stmts←END LOOP opt.id. ;
END  shift 276
.  error

```


state 243

```
iter$spec : FOR loop$para IN←opt.REVERSE. disc$range  
opt.REVERSE. : ← (154)  
REVERSE shift 278  
. reduce 154  
opt.REVERSE. goto 277
```

state 244

```
obj$decl : id$list : opt.CONSTANT. type←opt.$.expr. ;  
opt.$.expr. : ← (11)  
:= shift 257  
. reduce 11  
opt.$.expr. goto 279
```

state 245

```
type : type$mark← (17)  
. reduce 17
```

state 246

```
type$decl : TYPE id IS type$defn←;  
; shift 280  
. error
```

state 247

```
type$defn : enum$type$defn← (18)  
. reduce 18
```

state 248

```
type$defn : int$type$defn← (19)
```



```

        . reduce 19
state 249
    type$defn : array$type$defn←      (20)
        . reduce 20
state 250
    type$defn : record$type$defn←      (21)
        . reduce 21
state 251
    enum$type$defn : (←enum$lit , fst.enum$lit. )
    id shift 282
    char$str shift 283
        . error
    enum$lit goto 281
state 252
    int$type$defn : range$constr←      (34)
        . reduce 34
state 253
    array$type$defn : ARRAY←( index fst.$index. ) OF
type$mark ;
    ( shift 284
        . error
state 254
    record$type$defn : RECORD←comp$list END RECORD
    fst.obj$decl. : ←      (55)

```



```

        . reduce 55
comp$list goto 285
fst.obj$decl. goto 286
state 255
        mod$body : mod$nature BODY id IS declarative$part
END id ;←      (197)
        . reduce 197
state 256
        para$decl : id$list : mode type$mark opt.$expr.←
(186)
        . reduce 186
state 257
        opt.$expr. : :=←expr
        opt.unary$op. : ←      (81)
NOT shift 53
+ shift 51
- shift 52
        . reduce 81
expr goto 287
simple$expr goto 48
rel goto 47
opt.unary$op. goto 49
unary$op goto 50
state 258

```



```

    range$constr : RANGE range+      (25)
    . reduce 25

state 259
    range : range$id .. simple$expr+  (26)
    . reduce 26

state 260
    fst.$.comp$assoc. : fst.$.comp$assoc. , comp$assoc+
(42)
    . reduce 42

state 261
    fst.$.choice. : fst.$.choice. ; choice+  (45)
    . reduce 45

state 262
    choice : range$id+      (52)
    . reduce 52

state 263
    choice : restr$choice+  (53)
    . reduce 53

state 264
    comp$assoc : restr$choice fst.$.choice. => expr+
(48)
    expr : expr+log$op rel
    AND shift 69
    OR shift 70

```



```

XOR  shift 71
.  reduce 48
log$op  goto 68

state 265
    opt..1st.$.choice..$.expr. :  fst.$.choice. => expr+
(47)

    expr :  expr+log$op rel
    AND  shift 69
    OR   shift 70
    XOR  shift 71
    .  reduce 47
    log$op  goto 68

state 266
    stmt :  << id >> stmt+      (117)
    .  reduce 117

state 267
    assignment$stmt :  var := expr ;+      (129)
    .  reduce 129

state 268
    assignment$stmt :  name := expr ;+      (130)
    .  reduce 130

state 269
    exit$stmt :  EXIT opt.id. opt.WHEN.cond. ;+      (161)
    .  reduce 161

```


state 270

opt.WHEN.cond. : WHEN cond← (160)

. reduce 160

state 271

fst.cond\$ext. : fst.cond\$ext. cond\$ext← (141)

. reduce 141

state 272

cond\$ext : AND.THEN←expr

opt.unary\$op. : ← (81)

NOT shift 53

+ shift 51

- shift 52

. reduce 81

expr goto 288

simple\$expr goto 48

rel goto 47

opt.unary\$op. goto 49

unary\$op goto 50

state 273

cond\$ext : OR.ELSE←expr

opt.unary\$op. : ← (81)

NOT shift 53

+ shift 51

- shift 52


```

.   reduce 81
expr goto 289
simple$expr goto 48
rel goto 47
opt.unary$op. goto 49
unary$op goto 50

state 274
      if$stmt      :      IF      cond      THEN
seq$of$stmts←fst.ELIF.cond.THEN.seq$of$stmts.
opt.ELSE.seq$of$stmts. END IF ;
      fst.ELIF.cond.THEN.seq$of$stmts. : ←      (135)
.   reduce 135
      fst.ELIF.cond.THEN.seq$of$stmts. goto 290

state 275
      fst.WHEN.choice..1st.$choice..$.seq$of$stmts.      :
fst.WHEN.choice..1st.$choice..$.seq$of$stmts.←WHEN choice
fst.$choice. => seq$of$stmts
      case$stmt      :      CASE      expr      OF
fst.WHEN.choice..1st.$choice..$.seq$of$stmts.←END CASE
      END shift 292
      WHEN shift 291
.   error

state 276
      basic$loop : LOOP seq$of$stmts END←LOOP opt.id. ;

```


LOOP shift 293

. error

state 277

iter\$spec : FOR loop\$para IN opt.REVERSE.+disc\$range

id shift 21

. error

type\$mark goto 125

name goto 107

disc\$range goto 294

subprog\$array\$var goto 22

selected\$comp goto 23

predefined\$attri goto 24

state 278

opt.REVERSE. : REVERSE← (155)

. reduce 155

state 279

obj\$decl : id\$list : opt.CONSTANT. type opt.\$expr.←;

; shift 295

. error

state 280

type\$decl : TYPE id IS type\$defn ;← (24)

. reduce 24

state 281

enum\$type\$defn : (enum\$lit←, fst.enum\$lit.)


```

        , shift 296
        . error

state 282
    enum$lit : id←      (32)
    . reduce 32

state 283
    enum$lit : char$str←  (33)
    . reduce 33

state 284
    array$type$defn : ARRAY (←index fst.$index. ) OF
type$mark ;
    id shift 21
    . error
    type$mark goto 299
    name goto 107
    index goto 297
    disc$range goto 298
    subprog$array$var goto 22
    selected$comp goto 23
    predefined$attri goto 24

state 285
    record$type$defn : RECORD comp$list←END RECORD
    END shift 300
    . error

```


state 286

```
fst.obj$decl. : fst.obj$decl.←obj$decl
comp$list : fst.obj$decl.← (57)
id shift 66
. reduce 57
obj$decl goto 301
id$list goto 100
```

state 287

```
opt.$expr. : := expr← (12)
expr : expr←log$op rel
AND shift 69
OR shift 70
XOR shift 71
. reduce 12
log$op goto 68
```

state 288

```
expr : expr←log$op rel
cond$ext : AND.THEN expr← (143)
AND shift 69
OR shift 70
XOR shift 71
. reduce 143
log$op goto 68
```

state 289


```

    expr :  expr←log$op rel
    cond$ext :  OR.ELSE expr←      (144)
    AND  shift 69
    OR  shift 70
    XOR  shift 71
    .  reduce 144
    log$op  goto 68

```

state 290

```

    fst.ELSIF.cond.THEN.seq$of$stmts.      :
fst.ELSIF.cond.THEN.seq$of$stmts.←ELSIF      cond      THEN
seq$of$stmts
    if$stmt      :      IF      cond      THEN      seq$of$stmts
fst.ELSIF.cond.THEN.seq$of$stmts.←opt.ELSE.seq$of$stmts. END
IF ;

```

```

    opt.ELSE.seq$of$stmts. : ←      (137)
    ELSE  shift 304
    ELSIF  shift 302
    .  reduce 137
    opt.ELSE.seq$of$stmts.  goto 303

```

state 291

```

    fst.WHEN.choice..1st.$choice..$.seq$of$stmts.      :
fst.WHEN.choice..1st.$choice..$.seq$of$stmts.  WHEN←choice
fst.$choice. => seq$of$stmts
    OTHERS  shift 123

```



```

id shift 171
num shift 89
char$str shift 90
. error
type$mark goto 125
name goto 107
range$id goto 262
lit goto 172
disc$range goto 122
choice goto 305
restr$choice goto 263
subprog$array$var goto 22
selected$comp goto 23
predefined$attri goto 24

state 292
case$stmt : CASE expr OF
fst.WHEN.choice..1st.$choice..$.seq$of$stmts. END←CASE
CASE shift 306
. error

state 293
basic$loop : LOOP seq$of$stmts END LOOP←opt.id. ;
opt.id. : ← (151)
id shift 104
. reduce 151

```



```

        opt.id. goto 307

state 294
        iter$spec      :      FOR      loop$para      IN      opt.REVERSE.
disc$range←      (156)
        . reduce 156

state 295
        obj$decl : id$list : opt.CONSTANT. type opt.$expr.
;←      (13)
        . reduce 13

state 296
        enum$type$defn : ( enum$lit ,←fst.enum$lit. )
fst.enum$lit. : ←      (29)
        . reduce 29
fst.enum$lit. goto 308

state 297
        array$type$defn : ARRAY ( index←fst.$index. ) OF
type$mark ;
        fst.$index. : ←      (35)
        . reduce 35
        fst.$index. goto 309

state 298
        index : disc$range←      (38)
        . reduce 38

state 299

```



```

    index : type$mark←      (39)

    disc$range : type$mark←RANGE range
    RANGE shift 186

    . reduce 39

state 300

    record$type$defn : RECORD comp$list END←RECORD
    RECORD shift 310

    . error

state 301

    fst.obj$decl. : fst.obj$decl. obj$decl←      (56)

    . reduce 56

state 302

    fst.ELIF.cond.THEN.seq$of$stmts.           :
fst.ELIF.cond.THEN.seq$of$stmts.           ELIF←cond      THEN
seq$of$stmts

    opt.unary$op. : ←      (81)
    NOT shift 53
    + shift 51
    - shift 52
    . reduce 81
    expr goto 199
    simple$expr goto 48
    rel goto 47
    opt.unary$op. goto 49

```



```

        unary$op goto 50

        cond goto 311

state 303
        if$stmt : IF cond THEN seq$of$stmts
fst.ELIF.cond.THEN.seq$of$stmts. opt.ELSE.seq$of$stmts.←END
IF ;

        END shift 312

        . error

state 304
        opt.ELSE.seq$of$stmts. : ELSE←seq$of$stmts
fst.stmt. : ← (112)
        . reduce 112
fst.stmt. goto 94
seq$of$stmts goto 313

state 305
        fst.WHEN.choice..1st.$.choice..$.seq$of$stmts. :
fst.WHEN.choice..1st.$.choice..$.seq$of$stmts. WHEN
choice←fst.$.choice. => seq$of$stmts
        fst.$.choice. : ← (44)
        . reduce 44
fst.$.choice. goto 314

state 306
        case$stmt : CASE expr OF
fst.WHEN.choice..1st.$.choice..$.seq$of$stmts. END CASE←

```


(147)

. reduce 147

state 307

basic\$loop : LOOP seq\$of\$stmts END LOOP opt.id.←;

; shift 315

. error

state 308

fst.enum\$lit. : fst.enum\$lit.←enum\$lit

enum\$type\$defn : (enum\$lit , fst.enum\$lit.←)

id shift 282

char\$str shift 283

) shift 317

. error

enum\$lit goto 316

state 309

fst.\$index. : fst.\$index.←, index

array\$type\$defn : ARRAY (index fst.\$index.←) OF

type\$mark ;

) shift 319

, shift 318

. error

state 310

record\$type\$defn : RECORD comp\$list END RECORD←

(54)


```

        . reduce 54

state 311
        fst.ELIF.cond.THEN.seq$of$stmts.                :
fst.ELIF.cond.THEN.seq$of$stmts.                ELIF      cond←THEN
seq$of$stmts
        THEN shift 320
        . error

state 312
        if$stmt      :      IF      cond      THEN      seq$of$stmts
fst.ELIF.cond.THEN.seq$of$stmts.                opt.ELSE.seq$of$stmts.
END←IF ;
        IF shift 321
        . error

state 313
        opt.ELSE.seq$of$stmts. : ELSE seq$of$stmts←      (138)
        . reduce 138

state 314
        fst.$choice. : fst.$choice.←! choice
        fst.WHEN.choice..1st.$choice..$.seq$of$stmts.      :
fst.WHEN.choice..1st.$choice..$.seq$of$stmts. WHEN choice
fst.$choice.←=> seq$of$stmts
        => shift 322
        ! shift 225
        . error

```



```

state 315
    basic$loop : LOOP seq$of$stmts END LOOP opt.id. ;←
(153)
    . reduce 153
state 316
    fst.enum$lit. : fst.enum$lit. enum$lit← (30)
    . reduce 30
state 317
    enum$type$defn : ( enum$lit , fst.enum$lit. )←
(31)
    . reduce 31
state 318
    fst.$index. : fst.$index. ,←index
    id shift 21
    . error
    type$mark goto 299
    name goto 107
    index goto 323
    disc$range goto 298
    subprog$array$var goto 22
    selected$comp goto 23
    predefined$attri goto 24
state 319
    array$type$defn : ARRAY ( index fst.$index. )←OF

```



```

type$mark ;

      OF shift 324
      . error

state 320

      fst.ELIF.cond.THEN.seq$of$stmts.          :
fst.ELIF.cond.THEN.seq$of$stmts.          ELIF          cond
THEN←seq$of$stmts

      fst.stmt. : ←      (112)
      . reduce 112
      fst.stmt. goto 94
      seq$of$stmts goto 325

state 321

      if$stmt      :      IF      cond      THEN      seq$of$stmts
fst.ELIF.cond.THEN.seq$of$stmts. opt.ELSE.seq$of$stmts. END
IF+;

      ; shift 326
      . error

state 322

      fst.WHEN.choice..1st.$choice..$.seq$of$stmts.          :
fst.WHEN.choice..1st.$choice..$.seq$of$stmts. WHEN choice
fst.$choice. =>←seq$of$stmts

      fst.stmt. : ←      (112)
      . reduce 112
      fst.stmt. goto 94

```



```

    seq$of$stmts goto 327

state 323
    fst$.index. :  fst$.index. , index+      (36)
    . reduce 36

state 324
    array$type$defn :  ARRAY ( index  fst$.index. )
OF+type$mark ;
    id shift 21
    . error
    type$mark goto 328
    name goto 107
    subprog$array$var goto 22
    selected$comp goto 23
    predefined$attri goto 24

state 325
    fst.ELIF.cond.THEN.seq$of$stmts. :
fst.ELIF.cond.THEN.seq$of$stmts.      ELIF      cond      THEN
seq$of$stmts+      (136)
    . reduce 136

state 326
    if$stmt : IF cond THEN seq$of$stmts
fst.ELIF.cond.THEN.seq$of$stmts. opt.ELSE.seq$of$stmts. END
IF ;+      (139)
    . reduce 139

```


state 327

```
fst.WHEN.choice..1st.$.choice..$.seq$of$stmts.      :  
fst.WHEN.choice..1st.$.choice..$.seq$of$stmts.  WHEN choice  
fst.$.choice. => seq$of$stmts+      (146)  
      . reduce 146
```

state 328

```
array$type$defn : ARRAY ( index fst.$.index. ) OF  
type$mark+;  
      ; shift 329  
      . error
```

state 329

```
array$type$defn : ARRAY ( index fst.$.index. ) OF  
type$mark ;+      (37)  
      . reduce 37
```

71/95 terminals, 113/150 nonterminals

205/250 grammar rules, 330/475 states

0 shift/reduce, 0 reduce/reduce conflicts reported

113/150 working sets used

memory: states,etc. 2118/4000, parser 458/1500

230/250 distinct lookahead sets

404 extra closures

330 shift entries, 12 exceptions

193 goto entries

203 entries saved by goto default

Optimizer space used: input 935/4000, output 380/1500

380 table entries, 0 zero

maximum spread: 256, maximum offset: 324

APPENDIX D

Test Programs and Outputs

This Appendix illustrates the scanner/parser actions accomplished by this thesis. The test programs were written to include the syntactic constructs of Ada/MCS and to fully exercise the capabilities of the parser and scanner. The first four programs include the output of the scanner/parser in order to demonstrate the output format. The final six programs are provided with no output. The odd numbered programs are syntactically correct and each program is followed by a similar program with an error included.

Program one

FUNCTION sin IS

BEGIN

FOR x IN z RANGE 1..10

LOOP

x := a + b;

END LOOP;

cos(x,y);

RETURN x ;

END sin;

Output one

```
14 ; FUNCTION
    ; opt.vis$restriction.
    ; opt.SEPARATE.
    ; subprog$nature
54 ; sin
    ; designator
18 ; IS
    ; opt.formal$part.
    ; opt.RETURN.type$mark.
    ; subprog$spec
5 ; BEGIN
    ; opt.use$clause.
    ; fst.decl.
    ; fst.body.
    ; declarative$part
    ; fst.stmt.
13 ; FOR
54 ; x
    ; loop$para
17 ; IN
54 ; z
    ; opt.REVERSE.
```



```

    | name
29 | RANGE
    | type$mark
55 | 1
    | lit
    | range$id
71 | ..
55 | 10
    | opt.unary$op.
    | lit
    | pri
    | fac
    | fst.mult$op.fac.
19 | LOOP
    | term
    | fst.adding$op.term.
    | simple$expr
    | range
    | disc$range
    | iter$spec
    | opt.iter$spec.
    | fst.stmt.
54 | x
    | name

```



```

70 | :=
54 | a
    | opt.unary$op.
    | name
92 | +
    | pri
    | fac
    | fst.mult$op.fac.
    | term
    | fst.adding$op.term.
    | adding$op
54 | b
    | name
78 | ;
    | pri
    | fac
    | fst.mult$op.fac.
    | term
    | fst.adding$op.term.
    | simple$expr
    | opt.relal$op.simple$expr.
    | rel
    | expr
    | assignment$stmt

```



```

    | simple$stmt
    | stmt
    | fst.stmt.
11 | END
    | seq$of$stmts
19 | LOOP
78 | ;
    | opt.id.
    | basic$loop
    | loop$stmt
    | compound$stmt
    | stmt
    | fst.stmt.
54 | cos
    | name
79 | (
54 | x
    | opt.unary$op.
    | name
82 | ,
    | pri
    | fac
    | fst.mult$op.fac.
    | term

```



```

| fst.adding$op.term.
| simple$expr
| opt.relal$op.simple$expr.
| rel
| expr
| fst.$expr.
54 | y
| opt.unary$op.
| name
80 | )
| pri
| fac
| fst.mult$op.fac.
| term
| fst.adding$op.term.
| simple$expr
| opt.relal$op.simple$expr.
| rel
| expr
| fst.$expr.
| subprog$array$var
| name
78 | ;
| subprog$call$stmt

```



```

    | simple$stmt
    | stmt
    | fst.stmt.
32 | RETURN
54 | x
    | opt.unary$op.
    | name
78 | ;
    | pri
    | fac
    | fst.mult$op.fac.
    | term
    | fst.adding$op.term.
    | simple$expr
    | opt.relal$op.simple$expr.
    | rel
    | expr
    | opt.expr.
    | return$stmt
    | simple$stmt
    | stmt
    | fst.stmt.
11 | END
    | seq$of$stmts

```


54 | sin

78 | ;

| subprog\$body

| unit\$body

| compilation\$unit

Program two

FUNCTION sin IS

BEGIN

FOR x IN z RANGE 1..10

LOOP

x := a + b;

END LOOP;

cos(x,y);

RETURN x ;

END ? sin;

Output two

```
14 | FUNCTION
    | opt.vis$restriction.
    | opt.SEPARATE.
    | subprog$nature
54 | sin
    | designator
18 | IS
    | opt.formal$part.
    | opt.RETURN.type$mark.
    | subprog$spec
5 | BEGIN
    | opt.use$clause.
    | fst.decl.
    | fst.body.
    | declarative$part
    | fst.stmt.
13 | FOR
54 | x
    | loop$para
17 | IN
54 | z
    | opt.REVERSE.
```



```

    | name
29 | RANGE
    | type$mark
55 | 1
    | lit
    | range$id
71 | ..
55 | 10
    | opt.unary$op.
    | lit
    | pri
    | fac
    | fst.mult$op.fac.
19 | LOOP
    | term
    | fst.adding$op.term.
    | simple$expr
    | range
    | disc$range
    | iter$spec
    | opt.iter$spec.
    | fst.stmt.
54 | x
    | name

```



```

70 | :=
54 | a
    | opt.unary$op.
    | name
92 | +
    | pri
    | fac
    | fst.mult$op.fac.
    | term
    | fst.adding$op.term.
    | adding$op
54 | b
    | name
78 | ;
    | pri
    | fac
    | fst.mult$op.fac.
    | term
    | fst.adding$op.term.
    | simple$expr
    | opt.relat$op.simple$expr.
    | rel
    | expr
    | assignment$stmt

```



```

    | simple$stmt
    | stmt
    | fst.stmt.
11 | END
    | seq$of$stmts
19 | LOOP
78 | ;
    | opt.id.
    | basic$loop
    | loop$stmt
    | compound$stmt
    | stmt
    | fst.stmt.
54 | cos
    | name
79 | (
54 | x
    | opt.unary$op.
    | name
82 | ,
    | pri
    | fac
    | fst.mult$op.fac.
    | term

```



```

: fst.adding$op.term.
: simple$expr
: opt.relal$op.simple$expr.
: rel
: expr
: fst.$expr.
54 : y
: opt.unary$op.
: name
80 : )
: pri
: fac
: fst.mult$op.fac.
: term
: fst.adding$op.term.
: simple$expr
: opt.relal$op.simple$expr.
: rel
: expr
: fst.$expr.
: subprog$array$var
: name
78 : ;
: subprog$call$stmt

```



```

    | simple$stmt
    | stmt
    | fst.stmt.
32 | RETURN
54 | x
    | opt.unary$op.
    | name
78 | ;
    | pri
    | fac
    | fst.mult$op.fac.
    | term
    | fst.adding$op.term.
    | simple$expr
    | opt.relat$op.simple$expr.
    | rel
    | expr
    | opt.expr.
    | return$stmt
    | simple$stmt
    | stmt
    | fst.stmt.
11 | END
    | seq$of$stmts

```


Scanner error:unknown symbol

54 ! sin

78 ! ;

! subprog\$body

! unit\$body

! compilation\$unit

Program three

```
PACKAGE test IS
    TYPE s IS RANGE 1 .. 10 ;
    TYPE t IS RECORD
        idfer, idfer1, idfer2 : s;
    END RECORD;
    id1 : t;
    TYPE set IS ARRAY(red,yellow,blue) OF colors;
    primary : set;
    TYPE victor IS (whiskey,foxtrot);
END test
```


Output three

```
27 | PACKAGE
    | opt.vis$restriction.
    | opt.SEPARATE.
    | mod$nature
54 | test
18 | IS
36 | TYPE
    | opt.use$clause.
    | fst.decl.
54 | s
18 | IS
29 | RANGE
55 | 1
    | lit
    | range$id
71 | ..
55 | 10
    | opt.unary$op.
    | lit
    | pri
    | fac
    | fst.mult$op.fac.
```



```

78 | ;
    | term
    | fst.adding$op.term.
    | simple$expr
    | range
    | range$constr
    | int$type$defn
    | type$defn
    | type$decl
    | decl
    | fst.decl.
36 | TYPE
54 | t
18 | IS
30 | RECORD
    | fst.obj$decl.
54 | idfer
    | fst.$id.
82 | ,
54 | idfer1
    | fst.$id.
82 | ,
54 | idfer2
    | fst.$id.

```



```

91 | :
    | id$list
54 | s
    | opt.CONSTANT.
    | name
78 | ;
    | type$mark
    | type
    | opt.$expr.
    | obj$decl
    | fst.obj$decl.
11 | END
    | comp$list
30 | RECORD
    | record$type$defn
    | type$defn
78 | ;
    | type$decl
    | decl
    | fst.decl.
54 | id1
    | fst.$id.
91 | :
    | id$list

```



```

54 | t
    | opt.CONSTANT.
    | name
78 | ;
    | type$mark
    | type
    | opt.$expr.
    | obj$decl
    | decl
    | fst.decl.
36 | TYPE
54 | set
18 | IS
3  | ARRAY
79 | (
54 | red
    | name
82 | ,
    | type$mark
    | index
    | fst.$index.
54 | yellow
    | name
82 | ,

```



```

    | type$mark
    | index
    | fst.$.index.
54 | blue
    | name
80 | )
    | type$mark
    | index
    | fst.$.index.
23 | OF
54 | colors
    | name
78 | ;
    | type$mark
    | array$type$defn
    | type$defn
    | type$decl
    | decl
    | fst.decl.
54 | primary
    | fst.$.id.
91 | :
    | id$list
54 | set

```



```

    | opt.CONSTANT.
    | name
78 | ;

    | type$mark
    | type
    | opt.$expr.
    | obj$decl
    | decl
    | fst.decl.
36 | TYPE
54 | victor
18 | IS
79 | (
54 | whiskey
    | enum$lit
82 | ,
    | fst.enum$lit.
54 | foxtrot
    | enum$lit
    | fst.enum$lit.
80 | )
    | enum$type$defn
    | type$defn
78 | ;

```



```

    | type$decl
    | decl
    | fst.decl.
11 | END
    | fst.body.
    | declarative$part
    | opt.IS.declarative$part.
54 | test
    | opt.id.
    | mod$spec
    | unit$body
    | compilation$unit
```


Program four

```
PACKAGE test IS
    TYPE s IS RANGE 1 .. 10 ;
    TYPE t IS RECORD
        idfer, idfer1, idfer2 : s;
    END RECORD;
    id1 : t;
    TYPE set IS ARRAY(red,yellow,blue) OF colors;
    primary : set;
    TYPE victor IS (whiskey,foxtrot);
END test ;
```


Output four

```
27 ; PACKAGE
    ; opt.vis$restriction.
    ; opt.SEPARATE.
    ; mod$nature
54 ; test
18 ; IS
36 ; TYPE
    ; opt.use$clause.
    ; fst.decl.
54 ; s
18 ; IS
29 ; RANGE
55 ; 1
    ; lit
    ; range$id
71 ; ..
55 ; 10
    ; opt.unary$op.
    ; lit
    ; pri
    ; fac
    ; fst.mult$op.fac.
```



```

78 ; ;
    ; term
    ; fst.adding$op.term.
    ; simple$exor
    ; range
    ; range$constr
    ; int$type$defn
    ; type$defn
    ; type$decl
    ; decl
    ; fst.decl.

```

```

36 ; TYPE

```

```

54 ; t

```

```

18 ; IS

```

```

30 ; RECORD

```

```

    ; fst.obj$decl.

```

```

54 ; idfer

```

```

    ; fst.$id.

```

```

82 ; ,

```

```

54 ; idfer1

```

```

    ; fst.$id.

```

```

82 ; ,

```

```

54 ; idfer2

```

```

    ; fst.$id.

```



```

91 | :
    | id$list
54 | s
    | opt.CONSTANT.
    | name
78 | ;
    | type$mark
    | type
    | opt.$expr.
    | obj$decl
    | fst.obj$decl.
11 | END
    | comp$list
30 | RECORD
    | record$type$defn
    | type$defn
78 | ;
    | type$decl
    | decl
    | fst.decl.
54 | id1
    | fst.$id.
91 | :
    | id$list

```



```

54 | t
    | opt.CONSTANT.
    | name
78 | ;
    | type$mark
    | type
    | opt.$expr.
    | obj$decl
    | decl
    | fst.decl.
36 | TYPE
54 | set
18 | IS
3  | ARRAY
79 | (
54 | red
    | name
82 | ,
    | type$mark
    | index
    | fst.$index.
54 | yellow
    | name
82 | ,

```



```

    | type$mark
    | index
    | fst.$.index.
54 | blue
    | name
80 | )
    | type$mark
    | index
    | fst.$.index.
23 | OF
54 | colors
    | name
78 | ;
    | type$mark
    | array$type$defn
    | type$defn
    | type$decl
    | decl
    | fst.decl.
54 | primary
    | fst.$.id.
91 | :
    | id$list
54 | set

```



```

    | opt.CONSTANT.
    | name
78 | ;

    | type$mark
    | type
    | opt.$expr.
    | obj$decl
    | decl
    | fst.decl.
36 | TYPE
54 | victor
18 | IS
79 | (
54 | whiskey
    | enum$lit
82 | ,
    | fst.enum$lit.
54 | foxtrot
    | enum$lit
    | fst.enum$lit.
80 | )
    | enum$type$defn
    | type$defn
78 | ;

```



```

      | type$decl
      | decl
      | fst.decl.
11 | END
      | fst.body.
      | declarative$part
      | opt.IS.declarative$part.
54 | test
      | opt.id.
      | mod$spec
      | unit$body
      | compilation$unit
78 | ;

```

syntax error

Program five

```
PROCEDURE test+it IS
```

```
  a,aborting, abo: integer;
```

```
  TYPE t IS ARRAY(x RANGE 1..10, y RANGE 1..3) OF float;
```

```
  amatrix : t;
```

```
BEGIN
```

```
  a := 16#3ab7;
```

```
  b := "this is a character string."; -- this is a comment!!!!
```

```
  abo := 6.5E-4;
```

```
END test+it ;
```


Program six

```
PROCEDURE test←it IS
```

```
  a,aborting, abo: integer;
```

```
  TYPE t IS array(x RANGE 1..10, y RANGE 1..3) OF float;
```

```
  amatrix : t;
```

```
BEGIN
```

```
  a := 16#3ab7;
```

```
  b := "this is a character string."; -- this is a comment!!!!
```

```
  abo := 6.5E-4;
```

```
END test←it ;
```


Program seven

```
PROCEDURE idfer IS
    TYPE t IS ARRAY(subscript RANGE number..number) OF idfer;
    idfer, idfer : idfer;
    idfer : CONSTANT t;

BEGIN
    idfer := number * number MOD idfer;
    LOOP
        IF number THEN
            idfer := number;
        END IF;
        EXIT WHEN (number);
    idfer := char+string;    -- comment
    END LOOP;
END idfer ;
```


Program eight

PROCEDURE idfer IS

 TYPE t IS ARRAY(subscript RANGE number..number) OF idfer;

 idfer, idfer : idfer;

 idfer : CONSTANT t;

BEGIN

 idfer := number * number MOD idfer;

 LOOP

 IF number THEN

 idfer := number;

 END IF

 EXIT WHEN (number);

 idfer :? char←string; -- comment

 END LOOP;

END idfer ;

Program nine

PROCEDURE test IS

 idfer, idfer1, idfer2 : s;

BEGIN

 CASE a < b OF

 WHEN number =>

 a := c + d / e ;

 END CASE;

END test ;

Program ten

PROCEDURE test IS

 idfer, idfer1, idfer2 : s;

BEGIN

 CASE a < b OF

 a := c + d / e ;

 END CASE;

END test ;

LIST OF REFERENCES

1. "Preliminary Ada Reference Manual", SIGPLAN Notices, June 1979.
2. "Reference Manual for the Ada Programming Language", Proposed Standard Document, U.S. Department of Defense, July 1980.
3. Aho, Alfred V. and Ullman, Jeffrey D., Principles of Compiler Design, Addison-Wesley Publishing Company, 1977.
4. Archer, J., "Ada/CS, An Instructional Subset of the Programming Language Ada", Technical Report TR79-395, Computer Science Department, Cornell University, Nov 1979.
5. Johnson, Stephan C., "YACC - Yet Another Compiler-Compiler", Bell Laboratories, Murry Hill, NJ, 1975.
6. Joy, William, "An Introduction to Display Editing with Vi", Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, April 1979.
7. Lesk, M. E. and E. Schmidt, "Lex - A Lexical Analyzer Generator", Bell Laboratories, Murry Hill, NJ, Oct 1979.
8. McCoy, Earl E. and Wetmore, Thomas III., "A Program for the Conversion of Productions in an Extended Backus-Naur-Form to an Equivalent Backus-Naur-Form", Report Number NPS52-80-010, Naval Postgraduate School, July 1980.
9. Ritchie, Dennis M. and Kernighan, Brian W., The C Programming Language, Prentice-Hall, Inc., 1978.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4. Asst Professor B.J. MacLennan, Code 52M1 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5. Professor Earl E. McCoy Department of Marketing Central Connecticut State College 1615 Stanley Street New Britain, Connecticut 06050	1
6. LCDR Mark A. Rogers, USN 103 Runnymede Lane Summerville, South Carolina 29483	2
7. LT Linda M. Myers, USN PO Box 147 Boonville, New York 13309	2

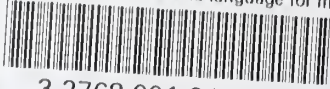
Thesis
R6854
c.1

Thesis 191153
R6854 Rogers
c.1 An adaptation of the
Ada language for ma-
chine generated com-
pilers.

15 JUL 82 26425
30 AUG 84 28563
17 OCT 84 29740
JUL 85 29820
AUG 85 30000

Thesis 191153
R6854 Rogers
c.1 An adaptation of the
Ada language for ma-
chine generated com-
pilers.

An adaptation of the Ada language for ma



3 2768 001 94966 2
DUDLEY KNOX LIBRARY